Thesis for the Degree of Doctor of Philosophy

A Scholarship Approach to Model-Driven Engineering

Håkan Burden

UNIVERSITY OF GOTHENBURG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Göteborg, Sweden 2014

A Scholarship Approach to Model-Driven Engineering
ISBN 978-91-628-9096-4

Technical Report no. 114D
Department of Computer Science and Engineering
Research groups: Language Technology and Model-Driven Engineering

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31–772 1000

Typeset with LaTeX using GNU Emacs
Printed at Chalmers University of Technology, Gothenburg, 2014

“It’s the side effects that save us”

The National

Abstract

Model-Driven Engineering is a paradigm for software engineering where software models are the primary artefacts throughout the software life-cycle. The aim is to define suitable representations and processes that enable precise and efficient specification, development and analysis of software.

Our contributions to Model-Driven Engineering are structured according to Boyer's four functions of academic activity – the scholarships of teaching, discovery, application and integration. The scholarships share a systematic approach towards seeking new insights and promoting progressive change. Even if the scholarships have their differences they are compatible so that theory, practice and teaching can strengthen each other.

Scholarship of Teaching While teaching Model-Driven Engineering to undergraduate students we introduced two changes to our course. The first change was to introduce a new modelling tool that enabled the execution of software models while the second change was to adapt pair lecturing to encourage the students to actively participate in developing models during lectures.

Scholarship of Discovery By using an existing technology for transforming models into source code we translated class diagrams and high-level action languages into natural language texts. The benefit of our approach is that the translations are applicable to a family of models while the texts are reusable across different low-level representations of the same model.

Scholarship of Application Raising the level of abstraction through models might seem a technical issue but our collaboration with industry details how the success of adopting Model-Driven Engineering depends on organisational and social factors as well as technical.

Scholarship of Integration Building on our insights from the scholarships above and a study at three large companies we show how Model-Driven Engineering empowers new user groups to become software developers but also how engineers can feel isolated due to poor tool support. Our contributions also detail how modelling enables a more agile development process as well as how the validation of models can be facilitated through text generation.

The four scholarships allow for different possibilities for insights and explore Model-Driven Engineering from diverse perspectives. As a consequence, we investigate the social, organisational and technological factors of Model-Driven Engineering but also examine the possibilities and challenges of Model-Driven Engineering across disciplines and scholarships.

Acknowledgments

I want to start by thanking my supervisors – Rogardt Heldal, Peter Ljunglöf and Tom Adawi. I am indebted to their inspiration and patience. I likewise want to acknowledge the various members of my PhD committee at Computer Science and Engineering – Aarne Ranta, Bengt Nordström, Robin Cooper, David Sands, Jan Jonsson, Koen Claessen and Gerardo Schneider. Thanks!

There are three research environments that I particularly want to mention. The first is the Swedish National Graduate School of Language Technology, GSLT, for funding my graduate studies. The second research environment is the Center for Language Technology at the University of Gothenburg, CLT, which has funded some of the traveling involved in presenting the publications included in the thesis. I was also very fortunate to receive a grant from the Ericsson Research Foundation that enabled me to present two of the publications.

Over the years I have met far too many people at conferences and workshops, in classrooms and landscapes to mention you all – I appreciate the talks we had in corridors and by the coffee machine. I have also had some outstanding room mates over the years. It's been a pleasure sharing office space with you. A special thank you to the technical and administrative staff who have made my academic strife so much easier.

There are some researchers and professionals in the outside world that deserve to be mentioned; Joakim Nivre, Leon Moonen, Toni Siljamäki, Martin Lundquist, Leon Starr, Stephen Mellor, Staffan Kjellberg, Jonn Lantz, Dag Sjøberg, Jon Whittle, Mark Rouncefield, John Hutchinson – as well as all my past and present students. Cheers to the engineers at Ericsson, Volvo Cars and Volvo Trucks that had so many insights to share and patience with my constant probing. And to (nearly) all anonymous reviewers – Thanks!

On the private side I want to thank Ellen, Malva, Vega, Tora and Björn for providing an alternative reality. And a big thanks to my numerous friends and relatives who keep asking me what I do for a living. To all my neighbors at Pennygången. Your encouragement and support has meant a lot.

Associates, Friends, Family, Relatives and Neighbours[1] – You've all helped me to become the scholar I want to be. Thank You!

Håkan Burden
Gothenburg, 2014

[1] Feel left out? I do hope I can compensate you with a free copy of my thesis!

Included Publications

Scholarship of Teaching

Håkan Burden, Rogardt Heldal, and Toni Siljamäki. Executable and Translatable UML - How Difficult Can it Be? In *Proceedings of APSEC 2011: 18th Asia-Pacific Software Engineering Conference*, Ho Chi Minh City, Vietnam, December 2011.

Håkan Burden, Rogardt Heldal, and Tom Adawi. Pair Lecturing to Model Modelling and Encourage Active Learning. In *Proceedings of ALE 2012, 11th Active Learning in Engineering Workshop*, Copenhagen, Denmark, June 2012.

Scholarship of Discovery

Håkan Burden and Rogardt Heldal. Natural Language Generation from Class Diagrams. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVa 2011*, Wellington, New Zealand, October 2011. ACM.

Håkan Burden and Rogardt Heldal. Translating Platform-Independent Code into Natural Language Texts. In *Proceedings of MODELSWARD 2013, 1st International Conference on Model-Driven Engineering and Software Development*, Barcelona, Spain, February 2013.

Scholarship of Application

Håkan Burden, Rogardt Heldal, and Martin Lundqvist. Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing. In *Proceedings of 6th International Workshop on Multi-Paradigm Modeling, MPM'12*, Innsbruck, Austria, October 2012. ACM.

Rogardt Heldal, Håkan Burden, and Martin Lundqvist. Limits of Model Transformations for Embedded Software. In *Proceedings of 35th Annual IEEE Software Engineering Workshop*, Heraklion, Greece, October 2012. IEEE.

Scholarship of Integration

Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial Adoption of Model-Driven Engineering - Are the Tools Really the Problem? In *Proceedings of MODELS 2013, 16th International Conference on Model Driven Engineering Languages and Systems*, Miami, USA, October 2013.

Ulf Eliasson and Håkan Burden. Extending Agile Practices in Automotive MDE. In *Proceedings of XM 2013, Extreme Modeling Workshop*, pages 11-19, Miami, USA, October 2013.

Håkan Burden, Rogardt Heldal, and Peter Ljunglöf. Enabling Interface Validation through Text Generation. In *Proceedings of VALID 2013, 5th International Conference on Advances in System Testing and Validation Lifecycle*, Venice, Italy, November 2013.

Håkan Burden, Rogardt Heldal, and Jon Whittle. Comparing and Contrasting Model-Driven Engineering at Three Large Companies. In *Proceedings of ESEM 2014, 8th International Symposium on Empirical Software Engineering and Measurement*, Torino, Italy, September 2014.

Additional Publications

The following publications are also the result of my time as a PhD student. The majority are peer-reviewed short papers or extended abstracts and therefor not included in the thesis. The exceptions are a technical report and a popular science contribution (as indicated when appropriate).

Håkan Burden, Rogardt Heldal, and Tom Adawi. Assessing individuals in team projects: A case study from computer science. Presented at *Conference on Teaching and Learning - KUL*, Gothenburg, Sweden, January 2011.

Håkan Burden, Rogardt Heldal, and Tom Adawi. Students' and teachers' views on fair grades - is it possible to reach a shared understanding? In proceedings of *3:e Utvecklingskonferensen för Sveriges ingenjörsutbildningar*, Norrköping, Sweden, 2011.

Håkan Burden. *Three Studies on Model Transformations - Parsing, Generation and Ease of Use*. Licentiate thesis, University of Gothenburg, Gothenburg, Sweden, June 2012.

Håkan Burden, Rogardt Heldal, and Tom Adawi. Pair Lecturing to Enhance Reflective Practice and Teacher Development. In *Proceedings of ISL2012 Improving Student Learning Symposium*, Lund, Sweden, August 2012.

Håkan Burden, Rogardt Heldal, and Tom Adawi. Pair Lecturing - Catching that teachable moment. Published in *Lärande i LTH* (A Swedish popular science magazine), October 2012.

Håkan Burden, Jones Belhaj, Magnus Bergqvist, Joakim Gross, Kristofer Hansson Aspman, Ali Issa, Kristoffer Morsing, and Qiushi Wang. An Evaluation of Post-processing Google Translations with Microsoft® Word. In *Proceedings of The Fourth Swedish Language Technology Conference*, Lund, Sweden, October 2012.

Håkan Burden, Rogardt Heldal, and Tom Adawi. Pair Lecturing Model-Driven Software Development. Presented at *Conference on Teaching and Learning - KUL*, Gothenburg, Sweden, January 2013.

Giuseppe Scanniello, Miroslaw Staron, Håkan Burden, and Rogardt Heldal. *Results from Two Controlled Experiments on the Effect of Using Requirement Diagrams on the Requirements Comprehension.* Technical report, Research Reports in Software Engineering and Management, Chalmers University of Technology and University of Gothenburg, March 2013.

Håkan Burden, Rogardt Heldal, and Peter Ljunglöf. Opportunities for Agile Documentation Using Natural Language Generation. In *Proceedings of ICSEA 2013, 8th International Conference on Software Engineering Advances*, Venice, Italy, November 2013.

Giuseppe Scanniello, Miroslaw Staron, Håkan Burden and Rogardt Heldal. On the Effect of Using SysML Requirement Diagrams to Comprehend Requirements: Results from Two Experiments. In *Proceedings of EASE 2014, 18th International Conference on Evaluation and Assessment in Software Engineering*, London, UK, May 2014.

Håkan Burden and Tom Adawi. Mastering Model-Driven Engineering. In *Proceedings of ITiICSE 2014, 19th Annual Conference on Innovation and Technology in Computer Science Education*, Uppsala, Sweden, June 2014.

Håkan Burden. Putting the Pieces Together - Technical, Organisational and Social Aspects of Language Integration for Complex Systems. In *Proceedings of GEMOC 2014, 2nd International Workshop on The Globalization of Modeling Languages*, Valencia, Spain, September 2014.

Håkan Burden, Sebastian Hansson and Yu Zhao. How MAD are we? Empirical Evidence for Model-driven Agile Development. In *Proceedings of XM 2014, 3rd Extreme Modeling Workshop*, Valencia, Spain, September 2014.

Imed Hammouda, Håkan Burden, Rogardt Heldal and Michel R.V. Chaudron. CASE Tools versus Pencil and Paper – A student's perspective on modeling software design. In *Proceedings of EduSymp 2014, 10th Educators' Symposium colocated with MODELS 2014*, Valencia, Spain, September 2014.

Personal Contribution

In the case of *Natural Language Generation from Class Diagrams*, *Translating Platform-Independent Code into Natural Language Texts* and *Enabling Interface Validation through Text Generation* the idea of natural language generation from software models was conceived by Peter Ljunglöf while I was the main contributor in planning and conducting the research. I was also the main author of the publications while Rogardt Heldal and Peter Ljunglöf helped in giving the contributions their context.

For *Pair Lecturing to Model Modelling and Encourage Active Learning* I developed the idea of pair lecturing together with Rogardt Heldal while Tom Adawi played a crucial role in helping us evaluate and communicate the results. I was the main author.

Rogardt Heldal initiated the introduction of Executable and Translatable UML into our course. I was a main contributor in terms of planning, conducting, evaluating and writing *Executable and Translatable UML - How Difficult Can it Be?*.

The two papers *Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing* and *Limits of Model Transformations for Embedded Software* were initiated and conducted without my contribution. Here my contribution was in evaluating and presenting the outcome together with Rogardt Heldal and Martin Lundqvist.

In the writing of *Industrial Adoption of Model-Driven Engineering - Are the Tools Really the Problem?* my contribution was in validating the taxonomy based on the interviews I had conducted as well as contributing to the writing of the publication in general and being the main author of section 5.

Rogardt Heldal and I came up with the idea for *Comparing and Contrasting Model-Driven Engineering at Three Large Companies* while Jon Whittle was instrumental in how to analyse and communicate the result. Jon Whittle and I wrote most of the text with Rogardt Heldal contributing to section 4. I am responsible for all data collection.

Ulf Eliasson was the main author of *Extending Agile Practices in Automotive MDE* except for section three to which we contributed equally. Since the publication is the synthesis of two independent studies both authors have a shared responsibility for data collection and evaluation.

Contents

Summary and Synthesis

Håkan Burden [1]
[1] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
2014

1 Introduction

1.1 Context – Raising the Level of Abstraction

In the beginning of software development the programs were manually specified by low-level instructions detailing how data at a specific memory location should be moved to a new location or how the data at one explicit location was the sum of the data at two other locations [1]. While the low-level instructions gave the programmer detailed expressions and full control it also meant that the programming was error prone, slow and updating the programs with new instructions was difficult. By automating some of the tasks – such as memory allocation – and combining repetitive sequences of low-level instructions to one high-level short-hand notation programming became more oriented towards the needs of the human users instead of the constraints of the machines.

In the early 1970's the trend towards more abstract representations had resulted in languages with so different ways of expressing the computations that they represented different paradigms. Examples of paradigms that stem from this era are imperative languages where C is still a major player, object-oriented languages such as Smalltalk and Java as well as functional languages like ML and Haskell. The evolution has continued and during the 1990's scripting languages like PHP and Perl saw the light.

1.2 Thesis Subject – Model-Driven Engineering

In continuation of the trend of raising the level of abstraction Model-Driven Engineering, MDE, emerged just after the turn of the century [65]. Models have been used for a long time in engineering disciplines, both to describe the present and to predict the future [100]. What distinguishes a model in the context of software development is that models are not just an aid to describe and predict – models can play a key role in implementation when used as programming languages [44]. By automatically transforming the abstract model into more concrete code MDE promises a means for handling the growing complexity of software as well as increasing the development speed and the quality of the software by automating tedious and error-prone tasks [80].

1.3 Thesis Contributions

This thesis reports on the possibilities and shortcomings of Model-Driven Engineering from three different settings, each with their own perspectives:

Education In order to improve our students' ability to develop models we introduced two changes to our course. The first change was to introduce a tool that enabled the models to be used as a programming language [27] and the second change was to introduce pair lecturing to encourage the students to actively participate in developing models during lectures [23].

Basic Research While models can be more abstract than the generated code it does not mean that they are necessarily easier to understand. We used existing technologies for transforming models into code to instead generate natural language

representations from class diagrams [21], high-level behavioural descriptions of the models [22] and component interfaces [24]. The benefit of this approach in comparison to generating from code is that the texts are reusable across different implementations of the same model.

Industrial Practice Raising the level of abstractions might seem to be a technical issue but our results enforce how the success of adopting Model-Driven Engineering depends on both organisational and social factors as well as technical [26, 54, 119]. In the long run Model-Driven Engineering empowers new user groups to become software developers as well as speeding up the development process [28, 39].

1.4 Conceptual Framework – Scholarship

From the onset the research on MDE was carried out independently within each perspective – improving how to teach MDE to students, exploring the possibilities of model transformations for natural language generation and the adaption of MDE in an industrial setting. During the latter half of our PhD studies we recognised that these three perspectives had common features and in 2013 we identified how they coincided with Boyer's notion of scholarship [13, 14].

Boyer refers to basic research as the scholarship of *discovery*, education maps to the scholarship of *teaching* while the scholarship of *application* covers all interaction with non-academic organisations including industry. Boyer also introduces a fourth scholarship named *integration* which aims to seek synergies among disciplines and scholarships as well as new interpretations of scientific contributions or open research questions.

While it was originally unintentional, our way of approaching MDE corresponds to Boyer's definition of Scholarship. We will therefore use the four scholarships of Discovery, Teaching, Application and Integration to structure the contributions of the included publications but also to show how the originally independent research tracks have merged and influenced each other over time.

1.5 Thesis Overview

The remaining sections of this chapter are organised as follows :

Summary and Synthesis The next section will give more detail to the concepts of Model-Driven Engineering and scholarship. The used research methodologies are then described in section 3. The contributions of the included publications are given per scholarship in section 4. Section 5 discusses insights from combining a scholarship approach and graduate studies in terms of opportunities for reflection. The conclusion is found in section 6 while future research directions are presented in section 7.

The publications supporting the contributions are then included as chapters for each scholarship:

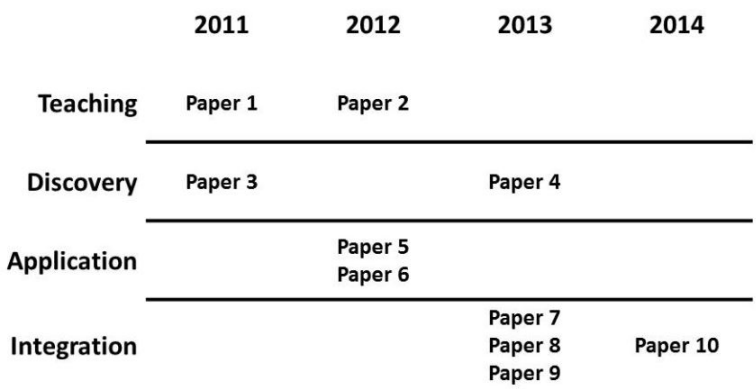

Figure 1: The included publications – referenced by paper – organised according to scholarship and publication year.

Appendix A: Scholarship of Teaching The relevant publications are included as paper 1 *Executable and Translatable UML – How Difficult Can it Be?* [27] and paper 2 *Pair Lecturing to Model Modelling and Encourage Active Learning* [23].

Appendix B: Scholarship of Discovery Included publications are paper 3 *Natural Language Generation from Class Diagrams* [21] and paper 4 *Translating Platform-Independent Code into Natural Language Texts* [22].

Appendix C: Scholarship of Application Paper 5 was published as *Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing* [26] and paper 6 as *Limits of Model Transformations for Embedded Software* [54].

Appendix D: Scholarship of Integration The four included papers are Paper 7 *Industrial Adoption of Model-Driven Engineering – Are the Tools Really the Problem?* [119], paper 8 *Extending Agile Practices in Automotive MDE* [39], paper 9 *Enabling Interface Validation through Text Generation* [24] and paper 10 *Comparing and Contrasting Model-Driven Engineering at Three Large Companies* [28].

An overview of the included papers, sorted under scholarship together with their respective publication year is found in Figure 1.

2 Background

Before we go further into the contributions of the included publications, the paradigm of Model-Driven Engineering and the concept of Scholarship are described in more detail.

2.1 Model-Driven Engineering

Model-driven engineering is a paradigm for developing software relying on models – or abstractions – of more concrete software representations to not only guide the development but constitute the single source of implementation.

2.1.1 Software Models

There are a number of definitions regarding the qualities and usage of models for scientific purposes in general [29, 35, 107] and also more specifically for software engineering [33, 72, 74, 94, 99, 100]. In the context of this thesis we will emphasise two notions of software models; first the relationship between model and code, second how models can be used as the actual implementation language(s).

Figure 2 gives the spectrum of model and code as defined by Brown [17]. Brown defines the code as a running application for a specific runtime platform (acknowledging that the code itself is a model of bit manipulations). In relation to Brown's definition of code, a model is defined as an artifact that assists in creating the code by transformations, enables prediction of software qualities and/or communicates key concepts of the code to various stakeholders.

Code only To the farthest left of Figure 2 is the code only approach to software development, using languages such as C or Java but without any separately defined models. While this approach often relies on abstractions such as modularization and APIs there is no notion of modelling besides partial analysis on whiteboards, paper and slide presentations. This scenario is referred to by Fowler as using models for sketching [44].

Code visualization A step to the right, and towards more modelling, is code visualization. Code is still the only implementation level but different kinds of abstractions are used for documentation and analysis. While some abstractions are developed manually some are automatically generated from the code – such as class diagrams or dependencies between components and modules.

Roundtrip engineering The third possibility according to Brown is that code and model together form the implementation. One example would be a system specification at model level that is then used to automatically generate a code skeleton fulfilling the specification. The skeleton is then manually elaborated to add more functionality and details. As the code evolves the model is updated with the new information to ensure consistency between specification and implementation. The term round-trip engineering comes from the fact that models and code are developed in parallel and require a roundtrip from model to model via code (or vice versa).

Model-centric In model-centric development the model specifies the software which is then automatically generated. In this case the models might include information about persistent data, processing constraints as well as domain-specific logic. In the words of Fowler, the modelling language becomes the programming

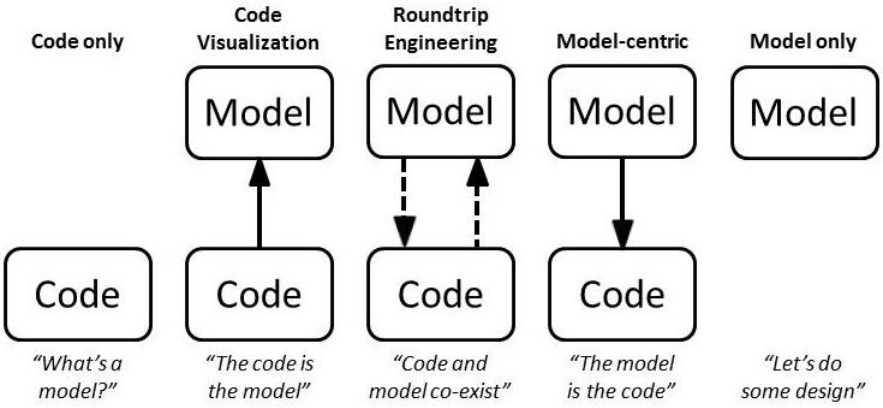

Figure 2: The software modelling spectrum as depicted by Brown [17].

language [44], analogously to how the source code becomes the machine code through code compilation [75].

Model only Finally, in the model only scenario the model is disconnected from its implementation. This is common in large organizations who traditionally are not software companies. After specifying the model it is out-sourced to suppliers and sub-contractors who either develop new software fitting the specification or deliver ready-made off-the-shelf solutions.

The spectrum given by Brown is of course a simplification, a model. Reality is not always that easy to map to the different scenarios and the scenarios can co-exist both as serial and parallel combinations. An example is software development at Volvo Cars. At system-level there is a model which specifies the overall system architecture. From this global model it is possible to generate partial models for specific subsystems. Some subsystems are then realised in-house using a round-trip engineering approach while other subsystems are developed using a code-centric approach where the partial model is seen as a blueprint [44] that the manually derived code must fulfill (a scenario not covered by Brown).

2.1.2 Executable Models

From the perspective of models as programming languages there are three important aspects to consider – the chosen representations of the modelling language, the possibility to deterministically interpret and *execute* a model and the ability to transform a source model to a target representation for a specific purpose.

Representations Over the decades there has been a strive towards raising the abstraction of programming languages [1]. First there were machine languages using 0's and 1's to represent wirings, then came assembly languages and then the third generation languages such as C and Java. For each generation the level

of abstraction rose and the productivity of the programmers followed [86]. The aim of using models as programming languages is to continue raising the level of abstraction and thereby also the productivity [73]. By reducing the number of concepts and/or the complexity of the representations the models should be easier to understand than the corresponding representations in C or Java. Exactly which representations that are chosen varies between the modelling languages, just as different third-generation languages have different representations and abstractions. A language where the chosen representations and the expressive power is restricted to a certain application domain is referred to as a Domain-Specific Language, DSL[2] [45, 77].

Executable Semantics An interpreter takes a source program and an input and returns an output [1]. In the case of models as programming languages the source program is a model which the interpreter can execute statement-by-statement in order to validate that the model has the right behaviour and structure [85]. In comparison to programming languages like Java and C, the modelling languages can consist of both graphical and textual constructions which in turn requires a more complex development environment to supply an interpreter that handles both textual and graphical elements as input and output.

Model Transformations In order to define a transformation between two (modelling) languages it is necessary to have a specification for each language. The common way of transforming a model into machine code is to first transform the model into a lower-level program language supported by a code compiler. In this way there is a need for an extra transformation when using a modelling language instead of a programming language such as C or Java [108]. It is also possible to reverse engineer a source into a more abstract target representation [76].

In this view the notion of code in Figure 2 can be either manually derived code or code automatically generated from a model. When the code is automatically generated from the model it is no longer a help in developing code – it becomes the single source of information and serves both as specification and implementation [75].

2.1.3 Model-Driven Approaches

The usage of models as the drivers for software development has many names where Model-Driven Architecture, MDA [17, 68, 75, 79], and Model-Driven Engineering, MDE [15, 65, 95], are among the most popular. The aim of adopting a model-driven approach instead of a code-centric is to handle the complexity of software development by raising the level of complexity and automate tedious or error-prone tasks [80].

MDA defines three abstraction layers and envisions that the development process transcends the layers from the more abstract to the most concrete.

[2]There is a discussion regarding when a DSL is a modelling language or a programming language – see for instance Yu et al. [114] – but we treat them as modelling languages since they from a user perspective abstract away from the low-level implementation details while introducing representations that capture the problem domain [60, 62].

Figure 3: The relationship between MDA, MDD and MDE given as a Venn diagram.

Computational-Independent Models The highest abstraction level in the MDA hierarchy uses the terminology of the domain to define the relevant functional properties. The structural relationship between the terms can also be defined using domain models [70] or taxonomies. It also defines the context – such as intended user groups, interacting systems and the physical environment – of the system while the system itself is treated as a black box, hence the name computational-independent. The computational-independent model is also referred to as a conceptual model [70] and abbreviated as CIM.

Platform-Independent Models At the next level of abstraction the Platform-Independent Models, PIM, introduces computational concepts such as persistence and safety together with interface specifications. The internal behaviour of the system might be defined by using executable modelling elements such as state machines or textual action languages that enhance the graphical elements [73, 85, 109]. The PIM should be possible to reuse across different platforms – combinations of operating systems, programming languages and hardware – by a model transformation that takes the PIM as an input and a specific platform setup as target. In relation to Figure 2 the PIM refers to the model.

Platform-Specific Models A platform-specific model, PSM, represents the lowest abstraction level and details what kind of memory storage should be used, which libraries and frameworks to include and if the system is to be run on one core or a distributed network etc. In relation to Brown's modelling spectrum a PSM is the code.

The concept of executable models is one way of realising the MDA vision since execution allows implementation and validation to be done at the PIM level while the PSM is obtained through automatic transformations [73, 75].

A taxonomy for distinguishing between different modelling approaches is given by Ameller [3] and depicted in Figure 3. Ameller distinguishes MDE from Model-Driven Development, MDD, by seeing MDE as more inclusive than MDD. The reason is that MDD represents a model-centric view and focuses on the transformation of abstract models into more concrete representations. According to Ameller (who cites Hutchinson et al. [60, 58]) MDE acknowledges the importance of a wider range of software engineering factors, including improved communication between stakeholders and quicker response towards changes, besides code generation. The difference between MDD and MDA lies in the fact that MDA only uses the standards conveyed by the

Object Management Group, OMG[3], such as the Unified Modeling Language, UML[4] [2, 12, 44], while MDD is a more pragmatic approach where you use what is most appropriate.

Ameller's distinction between MDA and MDD on the one hand and MDE on the other is consistent with the critique of MDA put forward by Kent [65]. Kent claims that MDA neglects social and organizational issues as well as the need for more differentiated levels of detail and abstraction than CIM, PIM and PSM. Furthermore, different kinds of abstractions are needed depending on the varying qualities of software and that in turn calls for the usage of domain-specific languages instead of generic representations.

From this point of view the distinction between MDE on the one hand and MDA and MDD on the other is that while MDE emphasises the model-centric view it also includes roundtrip engineering with the possibility of reverse engineering concrete representations into more abstract ones. MDA and MDD have a narrower scope and see the transformation from abstract to concrete as the fundamental relationship between model and code.

2.2 Scholarship

According to Boyer there are four functions that form the basis of academia and the roles of those it employs – the scholarships of discovery, teaching, application and integration [13, 14]. The foundation for all four functions is the systematic process of planning, executing, reflecting on and communicating innovation or changes to existing phenomena. Depending on the different functions of the four scholarships the nature of the overall process and the resulting insights will vary accordingly. Throughout the process the interchange with other scholars – such as discussions with peers and publishing – is instrumental to identify, and improve, the impact of the process. The reasoning from Boyer is that embracing the full functionality of the scholarship will remove the artificial conflict between teaching and research and instead practice, theory and teaching will interact and strengthen each other.

Figure 4 gives our interpretation of how the four functions represent different aspects of the scholarly process. For brevity we refer to each scholarship by its name where the initial letter has been capitalised, e.g Discovery refers to the scholarship of discovery.

Discovery The pursuit of new discoveries contributes in general to our knowledge and understanding of the world - more specifically discoveries are critical for a vital and intellectual climate within academia. Boyer stresses that not only is the discovery as such important, the systematic and disciplined process of conducting original research should also be stressed. The primary audience for new discoveries are academics within the same discipline while students and practitioners are also possible recipients and/or benefactors.

Teaching Just as for any of the other academic functions, teaching can and should be done in a systematic and disciplined way. As the terms scholarship of teaching

[3] `http://www.omg.org/` – Accessed 14 February 2014.
[4] `http://www.uml.org/` – Accessed 14 February 2014.

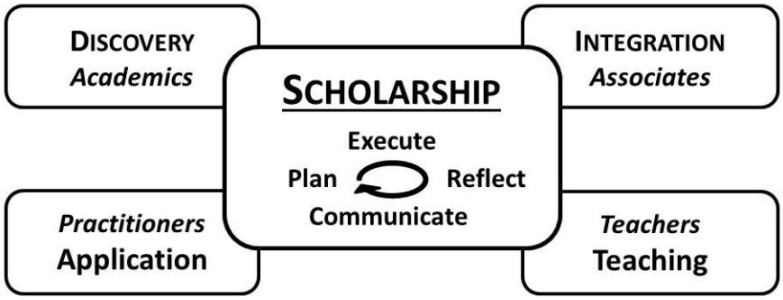

Figure 4: The four functions of SCHOLARSHIP with respective key *Scholars*.

suggests, Boyer expects the outcome to be communicated to other teachers first of all but the insights should serve the interests of the students either as shared theoretical insights or by the application of new insights on the teaching activities related to the discipline. The aim of teaching as a scholarship is to find new ways of encouraging active learning and critical thinking while providing a foundation for life-long learning. In this sense scholars are also learners where the interaction with the students is an opportunity to improve one's own understanding of the subject as well as the process of teaching and learning [57].

Application The scholarship of application refers to applying established discoveries to problems outside of academia. But it can also work in the opposite direction where the direction of future discovery is influenced by the challenges encountered in non-academic settings. In this way theory and practice interact and feed each other with new insights that will then be communicated to the practitioners (commercial or non-profit organizations, individuals or governmental bodies etc.) but also to academics within the discipline.

Integration The scholarship of integration seeks to illuminate established facts and original research from new perspectives or interpret them in new ways. As a consequence it opens up new disciplines or – using the terminology of Polanyi [83] – identifies over-laps among disciplines. Since integration is inter-disciplinary or even spans multiple scholarships, the assumption that the intended audience are knowledgable on all of the included topics becomes moot. Rather, the recipients will be specialists within some of the disciplines included in the integration, but not knowledgeable in all. This is a challenge not only in how to define a systematic methodology but also in how to communicate the results to reach across to all concerned parties[5]. Boyer also refers to integration as synergy or synthesis since it establishes connections among different disciplines or scholarships.

Boyer explicitly states that there is an overlap among the four and insights can fall into one or more scholarships while the intended audience can include more than one

[5]This thesis is an example of the challenges involved in communicating the research outcomes by integrating language technology, software engineering and engineering education research while applying MDE to disparate domains such as telecommunications and automotive software development.

kind of scholar [14]. An explicit example is how Teaching has elements of applying new discoveries from education and the subject area in a classroom context [113] or implicit examples such as knowledge transfer – applying academic discoveries in new organisations [40].

3 Research Methodologies

The included publications have been conducted using three different research methodologies – action research [64, 71], case studies [92, 121] and design science research [55, 103]. The research methodologies are not mutually exclusive, in fact they share the basic steps to *plan, execute, reflect* and *communicate*. The methodologies can also be combined – i.e. a case study can be applied within a design science research method to demonstrate the applicability of the design [82].

3.1 Action research

Action research is used to understand, but most of all improve, real-life situations in an iterative way [8, 47]. In relation to software engineering action research has been used mostly for implementing organizational changes related to software development – such as process improvement [61] and technology transfer [49] – in contrast to developing software artifacts [93]. Action research has also been advocated as a suitable research methodology for educational purposes. For instance, Stenhouse argues that action research can contribute both to the practice and theory of education and that communicating the resulting insights to other teachers is important in order to promote reflection within the community [111] (cited in [31]).

Action research is conducted over a series of iterations where each iteration can be broken down into three steps [36]:

Planning Defining the goals of the action, setting up the organization that will carry out the change and acquire the necessary permissions, knowledge and skills.

Acting Executing the plan and collecting the data that reflects the change. The data can consist of interviews, surveys, logs and/or observations but also documentation of the decisions that are made and their rationale.

Reflecting Evaluating the impact of the change in terms of collected data, discussing the organization and process with the participants, analysing the collected data as well as *communicating* the results to the relevant audiences.

A new iteration is initiated as new actions are identified during reflection. The detailed content and duration of each step varies depending on the objectives of the iteration and how the context has changed due to earlier interventions.

3.2 Case study

A case study aims to analyse one phenomenon or concept in its real-life context which in turn means that it is difficult to define the border between phenomenon and context

[10, 91, 121]. A question that is still under debate is if the findings from a case study can be generalised beyond the scope of the study or not, see e.g. [32, 43, 66].

According to Runeson et al. a case study can be broken down into five major steps which they claim can be used for all kinds of empirical studies [91]:

Case study design The design should specify the aim and the purpose of the study as well as lay out the overall plan.

Preparation for data collection The procedures and protocols for collecting the data relating to the objectives are defined.

Collecting evidence Depending on the design of the study the data can be collected from primary sources such as interviews and observations, indirect sources like monitoring tools or video recorded meetings but also data that already exists – e.g. requirements or source code – can be collected.

Analysis of collected data If the aim of the study is to explore a phenomena in its context the collected data is analysed inductively [91] – coding the data and then forming theories and hypotheses from the identified patterns among the codes. On the other hand the analysis can be deductive by its nature [91] and aim to validate or refute existing theories. In this case the collected data is compared to the predictions of the theories to be validated. Seaman argues that the analysis of the data should be carried out in parallel to the data collection since analysis can reveal the need for new data and allow the collection of data to confirm or refute emerging results [98].

Reporting The outcome of the study is reported to funders, industrial partners, educational communities etc. Different reports can be necessary depending on the intended audience.

Robson [89] states that case studies can be used to *explore, describe* or *explain* a phenomena in its context but case studies can also be used in an *emancipatory* way by changing aspects of the phenomena to empower certain stakeholders. This is in contrast to Runeson and Höst who argue that a case study is purely observational and that empirical studies that aim to change an existing situation should be referred to as action research [92].

In the case of *inductive* studies – where the aim is to generate new hypothesis and theories with respect to the investigated phenomena [91] – Seaman states that ideally further data collection can help to refute, validate or enrich emerging theories but this is not always possible [98]. Furthermore, Flyvbjerg argues that in the case of generating new insights it is more interesting to focus on the exceptional data than data commonly found throughout the study since this opens for new insights regarding the phenomena and re-interpretations of established discoveries [43].

3.3 Design Science Research

Design science research is a research methodology for creating software artifacts – designs – addressing problems encountered in research or practice [55, 82, 103]. In

this context Simon defines a design as a human-made or artificial object [103] and the aim of design science research is the systematic investigation of the design of computer systems used for computations, communication or information processing [56]. While different designs are driven by different needs – such as addressing problems raised in previous research or by practitioners – the outcome should not only be a new artifact but also a theoretical contribution to the relevant field of research [82].

According to Peffers et al. the design process can be described in six steps [82]:

1. Identify and motivate the problem and relate it to state-of-the art as formulated by previous research.

2. Define the objectives of the artifact that answers to the problem above. The objectives can be quantitative – such as runtime metrics or figures for precision and recall, or qualitative – like a description on how the artifact supports new solutions.

3. Design and develop the artifact so that it delivers a research contribution either in how it was implemented or in what it accomplishes.

4. Demonstrate the artifact on an instance of the original problem. Depending on the objectives the demonstration can be anything from a case study to a proof-of-concept implementation.

5. Evaluate the artifact in relation to the objectives and state-of-the-art. Iterate steps 3-5 until the artifact meets the objectives.

6. Communicate the outcome of the design, both in terms of the artifact as such but also the insights gained throughout the process.

Where action research, case studies and grounded theory emphasise the investigation of a phenomena in its context, design science research emphasises the development of an artifact. Thus the latter recognises the possibility to investigate designs outside of their original contexts [117].

4 Contributions

The included publications are described per scholarship, for each scholarship introducing the shared context of the publications and then the individual contributions of each publication. The first scholarship is Teaching since this is the origin for our investigations into software models and MDE. Discovery is next as it builds on the modelling technologies introduced for Teaching to explore the possibilities of natural language generation from software models. Then comes Application as we wanted to validate if our understanding of MDE had bearing in industry. Finally, insights from the three scholarships are pooled together in Integration where a project involving three large companies has a central role.

4.1 Scholarship of Teaching

The scholarship of Teaching is represented by two publications on improving how to teach MDE. The first publication, *Executable and Translatable UML – How Difficult Can it Be?* [27], is a case study on the introduction of executable models while the second publication, *Pair Lecturing to Model Modelling and Encourage Active Learning* [23], reports on the impact of pair lecturing MDE. The two papers are related in that they both address the challenge of transferring an understanding of the problem into a validated solution.

4.1.1 Mastering Modelling

In software development the process of understanding a problem and defining a solution using relevant abstractions involves an implicit form of modelling [46]. During this process the developer has as an internal model of the system which can be shared and discussed with others. Over time the developer's internal model evolves through the interactions with other stakeholders, the changing requirements and context of the system as well as through the validation of the implementation.

Cognitive Apprenticeship Mastering the process from understanding the problem to validating a solution is not necessarily an easy task. From the perspective of MDE education it is not obvious for all students how to share and discuss different solutions – and even harder to come up with them in the first place.

Cognitive apprenticeship [18] is an approach to instructional design that aims to teach how masters form their internal model and then step-by-step formalise it into a solution that fulfills the requirements. In this way the tacit knowledge as well as the alternative routes are made explicit so that novices – apprentices – learn from imitating and reflecting on the practice of a skilled master. The learning possibilities are structured as six teaching methods:

Modelling The master verbalises her own cognitive process of performing a task – including dead-ends – so that the novices can build their own cognitive model of the task.

Coaching The master observes the novice performing a task and offers feedback and hints in order to further develop the novices' ability.

Scaffolding Compared to scaffolding the novices get less support while performing a task in a setting which is assessed by the master to fit the abilities of the novice.

Articulation The novices are given the opportunity to articulate knowledge, reasoning skills as well as problem-solving strategies either in collaboration with another novices or in interaction with the master.

Reflection The novices compare their ability to that of other novices or the master but also to the emerging internal cognitive model of expertise.

Exploration The novice is given room to solve problems without the help of the master.

As teachers we need to facilitate a learning environment where students can explore MDE concepts in different contexts and validate the completeness of their models. These are challenges we have encountered in our own course on MDE.

Teaching Model-Driven Engineering The course runs over eight weeks and corresponds to 200 hours. Most of the time the students spend on a course project working on an open-ended problem. The first part of the project consists of developing blueprint CIMs of a hotel reservation system and then in the second part developing an executable PIM. The project work is supported by two lectures per week except the last week when the students demonstrate their projects. A majority of 90-130 students are at their third year in one of three different bachelor programs while some come from a master program in software engineering. During the evolution of the course we have aimed to address both how to capture domain knowledge using models and how to reason about software models as well as how to validate the functionality and structure of the models.

Cognitive apprenticeship was not originally part of the motivation of the first included paper [27] – the connection was made while synthesising the thesis. For the second included paper [23] the link was made during our reflection on the outcome while preparing for communicating the outcome to a broader audience.

4.1.2 *Paper 1:* How Difficult Can it Be?

The publication presents the outcome of letting our students develop executable PIMs as part of their project work [27]. Before we introduced executable models into our course the students spent 175 hours preparing models in our course and then 200 hours in the following course to manually transform them into Java. As a result it took months before they were able to test their design decisions. The models and the code rarely correlated and the different models were inconsistent with each other. And since the only way to validate the models was by manual model inspection the students often had a different view than the teachers on the level of detail in the model.

From an MDA perspective an executable PIM would be the remedy since the interpreter would give the students constant feedback [73, 85] as they explore how to transform their CIM into an executable PIM. Using the terminology of cognitive apprenticeship the supervision time was used for modelling and coaching while scaffolding and exploration would distinguish the work the teams did on their own. Hopefully the students would spend time on reflection on their own initiative but the demonstration at the end of the course was an opportunity for us as teachers to ensure they did reflect on their own ability.

The question we wanted to explore ourselves was to which extent it would be feasible to develop an executable PIM as part of the course?

During the first part of the project the teams consisted of eight students that split into two teams of four to develop executable models. Each team had a total of 300 hours to their disposal and we defined a set of evaluation criteria to specify the expected design and functionality. In the first year, 2009, there were 22 teams and in 2010 the number of students had increased and 28 teams of four participated. From

the evaluation in 2009 we learnt that the students wanted more tool supervision so in 2010 one of the earlier students spent 22 hours on helping out with tool related issues.

In 2009 all 22 teams managed to deliver an executable model within the time frame but four teams failed to meet the evaluation criteria. In 2010, 25 out of 28 teams delivered on time as well as following the criteria but two teams failed to deliver on time and one team did not meet the criteria. The 25 teams that succeeded to meet both time constraints and evaluation criteria had in general more elaborate models than the 18 teams of 2009. In 2010 we introduced a survey to get a more detailed picture of how much time the students had spent learning the tool and its modelling paradigm. Out of 108 students 90 responded and 75 of those estimated that they required up 40 hours before they were confident using the tool. Our own evaluation of the models gave that the quality in terms of details and consistency had improved in general and in some cases went beyond what we thought possible, given the context.

4.1.3 *Paper 2:* Pair Lecturing to Model Modelling

While the introduction of executable models changed the students project work to the better we still found that the students struggled to apply the lecture content to their project. Lectures tend to present a neat solution to a problem and in the case the process for obtaining the solution is given that is also given as a straight-forward process [112]. What we wanted to do was to increase the interaction with the students in order to adjust the lecture content to their needs since students learn more when they are actively involved during lectures [51, 84].

Pair lecturing lets us do just this. In the context of cognitive apprenticeship we used the lectures to model, articulate, reflect and explore different ways of modelling the domain of course registration. In this way we make explicit our own individual cognitive processes in interaction with the students where we let them be the masters of the domain and we are the masters of MDE. Together we explore different ways of modelling the same phenomena while discussing their pros and cons. The lectures become an arena for how to use models to organise the functionality and structure of the domain and how information from one kind of model can be passed on to other kinds of models and enriched with more details from the domain.

The publication aimed to answer two questions *Can pair lecturing encourage students to take a deep approach to learning in lectures?* and *What are the pros and cons of pair lecturing for students and teachers?* In order to answer the questions a survey was used to collect the impressions of the students – as assessments on a five-grade Likert scale and as comments in relation to four statements.

From the survey it was possible to conclude that pair lecturing enabled students to be more active during the lectures than what they were used to. One obvious way of for a student to be more active is to participate in the on-going discussion. However, in a lecture hall with 100 students it is difficult to get everyone involved in the discussion – both practically but also since not everyone wants to state their opinion in public. But by posing two solutions to one problem even the silent get more active since they need to chose one of the solutions to accommodate to their project or come up with their own. We also found that the students felt they could influence the lecture content more than usual which gave them more opportunities to address the issues of MDE

they found most challenging. The students also reported on remembering more after the lectures.

Regarding the pros and cons the students found that too many opinions could be confusing while they appreciated that complicated concepts were explained twice and in different ways by the two teachers. As teachers we found that we were out of comfort zone since we could not predict where the student interaction would take us. The most important change for us was the new possibility for *reflection-in-action* [96].

4.2 Scholarship of Discovery

This section will first relate the concept of natural language generation to model transformations before describing the contributions of the included publications – *Natural Language Generation from Class Diagrams* [21] and *Translating Platform-Independent Code into Natural Language Texts* [22]. While originally published as exploratory case studies it would have been more sound to label the investigations as design science research due to the fact that it is the transformations – the designs – that are in focus while the designs are motivated by the findings of Arlow et al. where the original context has been abstracted away. The validation of the technique is limited to the feasibility of generating natural language descriptions from software models – there is no validation of the usability of the texts in an industrial context.

4.2.1 Model Transformations

A model transformation is a set of rules that define how one or more constructs in a source language are to be mapped into one or more constructs of a target language [68] together with an algorithm that specifies how the rules are applied [75]. The rules are specified according to the syntactical specification of the source and target languages and since the specification is a model of the modelling language it is referred to as a meta-model [5, 67]. A translation is transformation where the source and target languages are defined by different meta-models [118].

Depending on the levels of abstractions of the source and target languages a translation has one of the following characteristics [76]:

Synthesis The translation from a more abstract source to a more concrete target language is referred to as a synthesis translation. An example is the translation from source code to machine code where the compiler synthesises the information in the source code with information about operating system and hardware [1].

Reverse Engineering To reverse engineer is the opposite of synthesising – the translation removes information present in the source when defining the target [30]. Brown's notion of code visualisation [17] is an example of reverse engineering.

Migration The source is migrated to a target when the languages have the same level of abstraction but different meta-models, e.g. when porting a program from one programming language to another [76].

Natural Language Generation from Software Models The publications realise the transformation from model to natural language in the same way, using a two-step process. The first step is to transform the relevant parts of the model into the equivalent constructs as defined by a linguistic model – in other words a grammar. The linguistic model is then used to generate the natural language text. The two transformations relate to Natural Language Generation which transforms computer-based representations of data into natural language text in a three-step process – text planning, sentence planning and linguistic realisation [9, 88]:

Text planning During text planning the data to be presented is selected and structured into the desired presentation order.

Sentence planning Then the individual words of the sentences are chosen and given their sequential order according to the syntactical structure of the target language.

Linguistic realisation Finally, each word is given the appropriate word form depending on case, tense and orthography.

Depending on the purpose of the target language, the resulting translation can either be an example of synthesis, reverse engineering or migration. I.e. a textual summary of the model would be a reverse engineering transformation, a migration translation would yield a text that exactly paraphrases the model and a translation that not only paraphrases the model but also adds contextual information would be a synthesis transformation.

Literate Modelling Both the included publications under Discovery stem from the same problem reported by Arlow et al. [4]. From their experiences of the aviation industry they claim that models are not always suitable for organising requirements into source code. Their example is based on the concept of code-sharing (one flight having more than one flight number) worth millions of pounds every year in revenue for the airlines. This is denoted by an asterisk (*) when transformed into a class diagram. They go on to claim that in a class diagram all requirements looks the same – making it impossible to distinguish which requirements are more important than others.

Arlow et al. further claim that in order to validate the correctness of a model it is necessary to understand the modelling paradigm (object-oriented in their case), knowledge of the used models to decipher the meaning of the different boxes and arrows as well as acquiring the necessary skills to use the different modelling tools that are used for producing and consuming the models [4].

The aim of the two publications is then to explore the possibilities to paraphrase the models as natural language texts – a medium stakeholders know how to consume [42]. The idea is that since all changes to a system are done at the PIM level the generated texts residing at the CIM level will be in sync with the generated PSM. Thus documentation and implementation are aligned with each other and there is a single source of information for both documentation and implementation.

4.2.2 *Paper 3:* Language Generation from Class Diagrams

For the first translation the input was class diagrams while the output paraphrased both the included classes and associations in a textual format. Any comments embedded in the diagram were taken at face value and appended to the corresponding text element. In this way the underlying motivations of the diagram is carried over to the textual description.

Our transformation rules were inspired by the work done by Meziane et al. [78]. They generate natural language descriptions of class diagrams using WordNet [41], a wide-coverage linguistic resource which makes it useful for general applications but can limit the use for domain-specific tasks. In contrast our transformation reuses the terminology of the model to generate a domain-specific lexicon. Since the transformation rules are defined using the meta-model of the modelling language they are generic for all models that conform to the meta-model. In combination these two qualities enable the described approach for transformation from model to natural language text to be applied to any domain.

4.2.3 *Paper 4:* Translating Platform-Independent Code

Where the prior publication translated a class diagram that defines the structure of the software this publication takes an action language – a behavioural description of the model – as its input. The difference between an action language and a programming language compares to the difference between a programming language and assembly code – where the programming language abstracts away from the *hardware* platform the action language abstracts away from the *software* platform [75]. This means that when a programming language enables computations without knowing the number of registers or how the structure of the stack, an action language enables the addition of the ten first values of a set without specifying it as an iteration of a list or a increment over an array. As a consequence, according to Mellor et al. [75], just as the computations are reusable across hardware, the actions are reusable across software.

To our knowledge this was the first attempt of generating natural language from an action language. In comparison to previous work on generating natural language from code (e.g. see [105, 106, 87, 50]) our approach enables one transformation to be reused across many software platforms. This means that the generated texts can be reused independent if the behaviour is realised using C, Java or Python. Another benefit is that if the transformation is done at code-level the transformation has to sieve away the platform-specific details [105] as well as be re-implemented for each software platform.

4.3 Scholarship of Application

The contributions under Application consist of two publications that report on the possibilities – *Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing* [26] – and challenges – *Limits of Model Transformations for Embedded Software* [54] – of multi-paradigmatic modelling from a case study in industry.

4.3.1 Multi-Paradigmatic Modelling

The case study at Ericsson was a testbed for evaluating the possibilities of combining multiple domain-specific modelling languages for implementing the channel estimation of a 4G base station.

Domain-Specific Modelling In relation to modelling techniques being applied to more complex systems consisting of domains with different characteristics, the necessity to apply modelling languages with appropriate representations has risen as well. A platform-independent modelling language tailored for a specific domain is referred to as a domain-specific modelling language, DSML [63].

In this context a domain represents one subject matter with a set of well-defined concepts and characteristics [101] that cooperate to fulfill the interactions of the domain [85]. This definition of a domain is in line with what Giese *et al.* [48] call a horizontal decomposition and ensures the separation of concerns between the domains [37] as well as information hiding [81]. Furthermore, each domain can be realised as one or more software components as long as these are described by the same platform-independent modelling language [85].

The aim of a DSML is to bridge the gap between how domain concepts are represented and how they are implemented. Therefor a DSML consists of a set of representations relevant for the domain and one or more code generators that transform a program expressed by the DSML into code for a specific platform. DSML is another way to approach the challenges of validating models as expressed by Arlow et al. – instead of knowing how to map the modelling paradigm and the modelling concepts to the domain [4], the modelling languages uses a paradigm and representations relevant for the domain.

Multi-paradigmatic modelling is a research field that addresses the challenges that arise when using multiple DSMLs within the same system. Among the challenges are co-execution, integration of languages and transformation from multiple sources to a shared platform [52].

Channel Estimation for 4G The underlying case study for the two publications was a testbed for using two different DSMLs for implementing the channel estimation of the LTE-A uplink of a 4G telecommunication system at Ericsson AB. The system was demonstrated at the Mobile World Congress in Barcelona 2011. The requirements included unconditional real-time constraints for calculations on synchronous data and contextual determination of when signals should be sent and processed.

The system was analysed as consisting of two domains with their own characteristics and representations. The first domain was the signal processing domain which is signified by more or less fixed algorithms that filter, convert or otherwise calculate on incoming data while keeping a minimum interaction with external factors. The second domain was referred to as the control domain since it controlled the flow of execution and was responsible for interacting with the surrounding environment, determining which signals to send and process given that the context changed every millisecond.

Traditionally such a system would be realised using C, at least at Ericsson. The problem is that the result is a mix-up of language-dependent details, hardware opti-

misations and desired functionality. Instead a multi-paradigmatic modelling approach was tried out were the signal processing domain was implemented in a textual language without side effects that was embedded in functional language [6, 7]. The control domain was implemented in the same object-oriented modelling language that we had previously introduced in our course [73, 85].

The case study set out to answer four research questions:

1. To which extent is it possible to develop the signal processing and the flow of execution independently of each other?

2. How can the sub-solutions be integrated into one application running on the designated hardware?

3. Which are the key challenges to consider when generating imperative code from a functional modelling language?

4. To what extent is it possible to reuse existing model transformations for new hardware?

Questions 1 and 2 were answered in *Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing* while questions 3 and 4 were addressed in *Limits of Model Transformations for Embedded Software*.

4.3.2 *Paper 5:* Multi-Paradigmatic Modelling of Signal Processing

The development process relied on the possibility to validate the domains independently. For this to be possible it is important that the modelling languages are executable in their own right and that they later on can be translated into deployable code, after the models have been validated to have the right structure and behaviour. A benefit of the chosen approach was that we avoided the difficulties regarding tool and language integration recently reported on by Mohagheghi et al. [80] and Torchiano et al. [116].

Since the DSML used for the digital signal processing was embedded in a functional language the domain was independently developed and validated using the interpreter of the functional language. The control domain was implemented and validated using the interpreter that came with the executable modelling tool. In both cases the interaction with the other domain was stubbed, using dummy values for testing the signal processing and simulating the execution of signals in the control flow. Regarding the DSMLs appropriateness for the domains, the experts in respective domain stated that the chosen representations of respective language were not ideal since it was a challenge to map the domain concepts to the representations of the DSMLs.

The interfaces between the domains could be negotiated through Scrum meetings [11, 97] since the two teams were located in the same office landscape. Thus the interfaces could be detailed as the understanding of the domains grew during implementation instead of in advance. The implementation of the interface was then hand-coded since it required multi-core deployment which neither of the DSMLs had representations to handle. Full interaction of the two domains could not be done until after code generation.

4.3.3 *Paper 6:* Limits of Model Transformations

In the case of transforming a functional DSML to an imperative platform we encountered a problem that could not be solved under the time constraints. During the transformation new variables were introduced for intermediate results. Since the available memory on the chip was scarce all variables were limited to 16 bits of memory. However, in some cases the intermediate results needed 32 bits in order for the calculations to be precise enough. If all intermediate variables had been assigned 32 bits the generated code would not fit on the chip. Since the intermediate variables were introduced by the transformation they could not be marked in the models for special treatment during transformation [75]. In the end the only solution feasible within the time frame was to manually edit these variables in the generated code. If Multi-Paradigmatic Modelling is going to be successful it is necessary to pay more attention towards cross-paradigmatic transformations when the target platform comes with hard constraints on memory or processing capabilities.

In the case of the object-oriented DSML there already existed a model transformation proven to be efficient [102]. However, the transformation needed to be updated to accommodate for the new platform. The transformation expert did a first iteration and the results seemed promising when the decision was taken further up in the organisation that his services were needed better elsewhere. In the decision between finishing the update of the transformation themselves or to manually edit the generated code the engineers decided that it would be quicker to post-edit the generated code. From an organisational point-of-view the introduction of MDE can imply that new skills are needed. If the skills are scarce among the engineers the experts can quickly become bottlenecks in the development.

4.4 Scholarship of Integration

Integration consists of four publications with a shared origin in a study of MDE that we undertook at three companies. The publications are presented in the same order as they were published since the three first publications present more specific aspects of the data while the fourth presents a fuller analysis – which explains why it was finished last. Since the collected data was used and analysed in different ways for the four publications the analysis procedure and results will be presented under each publication.

First out is a publication in relation to *Industrial Adoption of Model-Driven Engineering – Are the Tools Really the Problem?* [119]. The paper integrates data from an earlier study by Hutchinson et al. [58, 59, 60, 120] with a subset of the interviews from our own MDE study in order to explore the impact of tooling on the adoption of MDE.

The second paper is *Extending Agile Practices in Automotive MDE* [39]. At the same time as our own study an internal and independent study was conducted at one of the involved companies. The included paper is the result from integrating the two studies in relation to the possibilities and challenges concerning agile development from an MDE perspective.

For the third paper we integrated the text generation techniques developed under

the scholarship of discovery with interview data from our MDE study to came up with *Enabling Interface Validation through Text Generation* [24].

Finally, the fourth paper shares the general analysis from our MDE study as *Comparing and Contrasting Model-Driven Engineering at Three Large Companies* [28]. The reported integration is a validation of Hutchinson et al.'s aforementioned research on industrial adoption of MDE, together with new insights specific to our own study.

4.4.1 A Study of MDE at Three Companies

The study was planned during the autumn of 2012 and then executed in the spring of 2013 with the final analysis and dissemination taking part during the autumn of 2013. There was a considerable overlap between execution and analysis which enabled new data to be collected during the analysis in order to follow up on emerging themes and validate spiring hypothesis [98]. The study was an international collaboration with Jon Whittle at Lancaster University. He was involved in an earlier study assessing the industrial application of MDE which included a survey with more than 400 respondents and 19 interviews at 18 companies representing 10 different industrial sectors [58, 59, 60, 120].

Motivation The aim of our study was to compliment the earlier study – which we refer to as Hutchinson et al. since John Hutchinson was the primary investigator – with a deeper and more narrow investigation. In this context we wanted to answer two questions, *What are the similarities and differences in the way that the three companies have adopted and applied MDE?* and *How does the nature of practice at the three companies agree or disagree with the findings from Hutchinson et al.?*

Context The study involved three companies; Ericsson AB, Volvo Cars Corporation and the Volvo Group. The three companies had in common that they were all in a transition into more agile software development while their experiences of MDE varied.

At Ericsson we collaborated with a unit developing software for radio-base stations. The unit has been involved in various MDE projects since the late 1980's and have primarily used UML for both documentation, specification and code generation in relation to their software development. We interviewed nine engineers, ranging from newly employed software developers, senior architects to managers. The interviews were conducted from January to March 2013.

Electronic Propulsion Systems is a new unit at Volvo Cars that develop software for electric and hybrid vehicles which started to use MDE in 2010. The interviewees were 12 all-in-all, mainly experts in physics or electric engineering encoding their knowledge in a modelling language that supports code generation. Just as for Ericsson the interviews took place during January to March 2013.

The Volvo Group is a code-driven organisation but there are pockets of MDE development, mainly at the top-level system planning with partial code generation. Due to the fact that the Volvo Group entered the project during the spring of 2013 and that there were fewer MDE practitioners we only interviewed four engineers. The interviewees represented Truck Technologies and had both managerial roles as well as

being developers or architects. The later entry date meant that the interviews were conducted between March and June 2013.

Data Collection The interviews were semi-structured and lasted for approximately 60 minutes each. A typical interview started off with general questions about personal experiences and attitude towards MDE as well as educational background and current role in the company. Depending on the outcome of the first questions the rest of the interview would go into more depth, exploring topics that struck the interviewer as most interesting or controversial. The interviews were audio recorded and then transcribed. The interviews were complemented with additional data collection in order to cross-verify the quality [91]. Besides observations we also used informal interactions at the coffee machine or lunch table since this allows for more spontaneous interaction and data collection from other practitioners than those interviewed [34]. Our findings were then presented at technical meetings to the engineers as well as at regular process meetings with company representatives where the findings were discussed and we received additional interpretations of the data. During the complimentary data collection field notes were used to document insights and impressions.

4.4.2 *Paper 7:* Are the Tools Really the Problem?

The immaturity of the tools is an often mentioned reason for why MDE adaption fails [69, 99, 116, 115]. But we also know from Kent [65] and the study by Hutchinson et al. that social and organisational aspects impact the adoption of MDE in industry [58, 60]. The questions we investigated in this publication was *To what extent poor tools hold back adoption of MDE?* and *What aspects – both organisational and technical – should be considered in the next generation of MDE tools?*

The taxonomy was induced from the interviews conducted by Hutchinson et al. [58, 60]. Each transcript was coded by two researchers at Lancaster University. The codes were then grouped into four themes – technical, external organisational, internal organisational and social factors – each factor with its own categories. We then used the data collected at Volvo Cars and Ericsson to validate the taxonomy by mapping tool related issues in the interviews to the taxonomy. Whenever there was any exceptional issue or it was unclear how to classify an issue this was noted down and discussed among the researchers.

The technical factors relate to issues such as specific functionality like code generation, the impact on quality and productivity as well as complexity of the tools and the modelling languages. Adapting tools to governmental or commercial standards or the cost of buying tools are examples of external organisational factors. Under internal organisational factors we grouped the need for new skills when adopting MDE, finding the right problem to start with and how to adapt the existing process with the possibilities enabled by the tools. Social factors identify among other things how engineers relate to the tools due to the (lack of) trust towards tool vendors and tools. The four factors are to a certain extent overlapping and it is possible to arrange the identified categories in other ways. We also expect new aspects to be identified as new issues are found in future studies.

The taxonomy can be used for evaluating existing tools or as a checklist when developing new tools but we also identify a number of aspects that warrant more attention from the MDE research community. We therefore propose that:

- tools should match the intended users instead of forcing people to adapt to the tools;
- while commercial tool vendors focus on code generation from low-level designs the research community could explore tools for creativity and high-level designs;
- more effort is needed in identifying which problems are suitable for MDE;
- researchers pay more attention to understand how processes and MDE tools can improve overall software development and
- open MDE communities are important since practitioners often feel isolated due to the lack of on-line forums where tool related issues can be discussed.

The last proposition was identified during the validation phase and an appropriate category should be included under social factors. We also found that a single tool issue often maps to more than one category in the taxonomy which shows how the factors interact in determining the success of MDE adoption.

Returning to the original questions of investigation we found that the technical aspects of the tools do not support those who try to adopt MDE but also that the organisational and social factors are just as important to consider.

4.4.3 *Paper 8:* Agile Practices in Automotive MDE

The automotive industry is traditionally oriented towards mechanical and electrical engineering using a rigorous process for planning, developing, integrating and finally testing the integrated software, hardware and mechanical parts in a prototype vehicle before going into mass production. Each step of the process is given firm start and end dates minimising the possible overlap between activities.

During the last twenty years the influence of software has grown exponentially so that a modern car contains 100 or more individual hardware units [19, 38]. The rigorous process used for mechanical and hardware development is a barrier for software development since each iteration lasts for approximately half a year with the consequence that software designs specified during the planning stage cannot be validated until several months later. One problematic area for software development is the interfaces between the top-level components since these are frozen at the planning stage. If the need for new signals or parameters is identified during implementation these changes cannot be included in the interface until next iteration.

At Volvo Cars software development teams that work within a top-level component have been successful in applying Simulink[6] for developing software for electronic

[6] A modelling language commonly used for control engineering tasks such as regulating water levels or state transitions of a system: http://www.mathworks.se/products/simulink/ – Accessed 14 February 2014.

propulsion – even though the team members have no training in software development. Instead the engineers have a background in electrical engineering or physics. An important aspect here is that the engineers have used Simulink, and the related MatLab[7] tool, during their university training. The resulting models are then transformed into code by a code generator. By building up a simulation environment it is also possible to validate the behaviour of the sub-system models before the software is integrated on a prototype vehicle. This allows for much shorter iterations between changes and shortcuts the necessity of specifying the software before development since the specifiers implement their own domain expertise.

Our publication sets out to answer the question *Which are the challenges and possibilities for a more agile software development process on a system level?*. And the way we did it was through two independent case studies.

The first study was launched internally by Cars in order to explore how the current process could become more agile. The second study was our own on MDE at three different companies. The first study used semi-structured interviews as the main source of data, interviewing eight software developers between May 2012 and April 2013. In the publication we narrow the number of interviewees at Cars in the second study to eight so that the two sets of informants are consistent in terms of background and responsibilities. Though the two interview sets were analysed independently of each other they came to the same conclusion – something that was recognised by one of the managers involved in facilitating both studies.

The identified challenge towards more agile development at system level was the freezing of the interfaces. The engineers reported on different work-arounds to overcome the problems such as defining extra signals or parameters that might not be used. As a result the interfaces are difficult to understand with regards to the actual behaviour and consume more memory and processing capabilities than necessary. These problems are dealt with but lead to more work later in the development process than originally planned for.

To identify possible solutions to the identified challenge a system architect with responsibility for the system-wide interfaces was interviewed. The perceived remedy was more MDE. The focus in the future should be on the teams developing the software and not on the planning stage in the development cycle. By extending the current tool that manages the interfaces to also execute consistency checks the teams could add new signals as they were needed and old signals that were never used could be discarded. The consistency check would ensure that the new signals were compliant to legacy signals in a manner similar to what Rumpe advocates [90].

4.4.4 *Paper 9:* Validation through Text Generation

As seen in the previous publication the interfaces are problematic when they are overloaded with unused signals – it becomes difficult to know which signals are used and if there is an intended order between the signals. The problem of understanding the interface implementation was not unique to Volvo Cars but encountered at the other companies as well. It seemed to be an opportunity to try to integrate the insights regarding model transformations from Discovery with a problem found in industry.

[7] http://www.mathworks.se/products/matlab/ – Accessed 14 February 2014.

The aim of the design was to explore if a text describing an interface can be generated from an existing test model. The reason for using a test model is that it should capture the desired functionality of the system behind the interface, describing in which sequence the signals are supposed to be sent and what the expected response is [53]. The problem is that it is a model and not necessarily understood by all stakeholders, hence a possibility for natural language generation.

The contribution consists of generic transformation rules that transform a test model into a natural language text. The text specifies the intended order between the signals and also filters out the signals that are not covered by the test model. In contrast to the earlier publications under Discovery this transformation does not rely on an intermediate grammar for linguistic realisation. Instead the concepts of the source language are mapped into holes of a template representing the target language. In this way the transformations are easier to manipulate since they do not require an understanding of the target language's linguistic paradigm. And as we saw under Application it is positive if we can keep the complexity of the transformations low.

While reported as a case study the publication is more appropriately defined as a design study since we due to time constraints could not evaluate the outcome in its original context. The lack of validation in the original context means that the question of what we will find when balancing the complexity of the transformation with the fluency of the generated text is still open. The same is also true for how useful and readable the generated texts are and what measures can be taken to improve them.

4.4.5 *Paper 10:* Model-Driven Engineering at Three Companies

For this publication the focus was two-fold – to induce new hypothesis from the data [98, 43] and validate the findings of Hutchinson et. al [58, 59, 60, 120]. The interviews were transcribed and each transcript was coded by at least two researchers. The codes were then grouped into themes where the most interesting themes were selected for publication. Each selected theme was validated both by triangulation of data and by letting company representatives confirm the outcome before we submitted the paper. After this analysis was finished we extracted the findings from Hutchinson et al. and for each finding we went through the transcripts to see if the finding was supported or refuted by our data.

The first reported insight is that MDE enables software development to be brought back in-house. This is important to shorten lead times between the decision to implement a feature and its actual integration on platform. A key factor here is that it also enables a more agile way of working since software development can be done by the domain experts that traditionally specify what suppliers and consultants are supposed to deliver. In this way the dependency on the knowledge of external actors is mitigated as well as the risk that they misinterpret the specification or do not deliver on time. It also enables the development of new features in an exploratory fashion which is not possible if the functionality has to be detailed before development. The change is possible since the domain experts can encode their knowledge in a suitable domain-relevant language that supports the specification to be transformed into code.

Hutchinson et al. flag for the possibility that the gain of adopting MDE can be counterbalanced by the cost for adapting the tools for the specific context [60]. We

found that large-scale MDE does not only require adapting single tools but rather requires the introduction of a MDE-specific infrastructure that compensates for the domain experts lack of software development skills. We use the term primary software to denote the software developed as part of the product to deliver – in this case a car – and the term secondary software for the software that supports the development of the primary software. Since the primary software is explorative to its nature and developed under agile forms it is not possible to define the secondary software in advance – its functionality and content will depend on the directions taken for the primary software.

We found that abstractions do not necessarily help in understanding complex software. One example is how a modelling tool enforces the user to open new windows for each level in the hierarchical structure of the software as well as for each subpart at that level. The consequence, according to one interviewee, was that you could never get the details of a part and its context at the same time which made it difficult to interpret the given model elements.

In most cases our study validates the findings of Hutchinson et al. The exceptions regard the engineers' sense of control – Hutchinson et al. found that code gurus feel they loose control while managers are risk-averse and avoid new technologies including MDE – and the lack of relevant MDE skills among practitioners. In contrast we found that the introduction of MDE in a new unit enabled the adaption of new tools, new processes and the employment of engineers that had prior experience of the chosen modelling language. And coders employed to develop secondary software saw no loss of control in their role.

While interfaces between top-level components was a problem at Volvo Cars the engineers at Ericsson had new opportunities to address the challenge due to the introduction of agile development. At Ericsson the developers get access to both ends of the interface and can change it according to the changing understanding of what needs to be communicated over the interface. The organisational impact is enabled by the software residing on one hardware unit and all software being developed in-house. As reported on in *Extending Agile Practices in Automotive MDE* [39] neither are the case at Volvo Cars so other ways of handling the challenge are needed.

In our publication regarding MDE tools [119] we claim that the MDE community needs a better way of determining when and how to introduce MDE in an organisation. From our data it seems that forming a new unit was a successful way since it allowed the engineers to adapt the experiences from other units to a new organisational context without the legacy in terms of existing culture and software.

5 Reflection

Smith defines reflection as the *"assessment of what is in relation to what might or should be and includes feedback designed to reduce the gap"* [104]. Reflection is a vital part of teaching and learning where new situations constantly arise for which we have not been specifically trained [16, 96]. But as we saw in Section 3 on research methodologies, reflection plays an important part in evaluating the decisions and outcomes of a research process. As Steeples points out, the involved researcher has the dual roles

to both drive and critically review the process [110].

The four scholarships allow for different possibilities for reflection, in what Schön refers to as reflection-in-action and reflection-on-action [96]. Reflection-on-action takes place before or after the execution of an activity, at the planning or evaluation stage. Reflection-in-action is in contrast done during the activity. By changing the context the possibilities for reflection change since the time frame for reflection-in-action varies from seconds in a class room or during interviews to days or weeks during the implementation of a design. And the ability to reflect in-action which pair lecturing provided was useful during the interviews since it was necessary to both reflect on how the situation evolved while being involved in the discussion.

Different roles have different perspectives, even within the scholarships. A teacher and a student will see different challenges and possibilities and bring their own assumptions into the equation. The same goes for an engineer and a manager who will have different experiences and perspectives on software development within the same organisation. By interacting with a number of different practitioners the ability to see new possible interpretations of a phenomena grows.

In relation to the changing perspectives the level of expertise of the academics vary, both in relation to MDE and the domain it is used for. In some cases the academics' knowledge of MDE in general is greater than the practitioners, as in the class room, while in other cases practitioners have more in-depth knowledge of their way of working with MDE and the domain, while the contribution of the academic instead is to provide new interpretations. And in some case the academic becomes the novice yet again.

Transferring between the different scholarships has also enabled new perspectives on the diverse activities within the scholarships. Sometimes this has lead to publications but other times the results are more informal in terms of feedback on how to improve or handle different aspects. The collaboration behind *Executable and Translatable UML – How Difficult Can it Be?* is an example of where the discussions have led to a formal result in terms of both course improvement and publication. Another example is how the feedback from our students on executable modelling tools is the reason why the challenges facing the engineers regarding interfaces was recognised. In general, being active in more than one scholarship gives new contexts to air and discuss recurring challenges that often are shared but have different impact depending on context.

The different audiences when communicating the results of the research means insights into different research traditions such as computer science's focus on implementing designs in comparison to inducing new insights from empirical data.

6 Conclusions

The contributions under Teaching relate to how novice modellers can be helped to master the process of organising domain knowledge into software solutions. This was achieved by transitioning from models as blueprints to using models as the programming language [27]. The second change was to move from lectures focused on describing the features of the models to how we reason when using the models [23]. While the

scope of the course is model-driven development the skills should be transferable to code-centric development since the focus is on achieving good designs and articulating the pros and cons of different solutions.

The main theoretical contribution for Discovery is how Natural Language Generation relates to the concept of model transformations [20]. The benefit of generating natural language texts from models instead of code is that the texts can be reused across implementations and describe the same structure and behaviour independent of how it is realised in terms of hardware, operative system and software representations. The technique is also domain-independent since the terminology of the model is reused for the textual descriptions.

Our contributions towards the Application of MDE stem from an industrial case study on multi-paradigmatic modelling within the telecoms domain. We saw how executable models allowed the independent development of distinct domains [26] while the subsequent transformations caused problems due to both the different paradigms of the source and target languages as well as from organisational perspective [54].

While raising the level of abstraction might seem a technical issue, our contributions enforce how the success of adopting Model-Driven Engineering in industry depends on both organisational and social factors as well as technical [119]. We have also shown how balancing the chosen modelling techniques with a suitable organisation empowers new user groups as software developers as well as speeds up the development process [28, 39].

Our findings also support the claims of Kent [65] and Ameller [3] that using models in software development is more than a technical question and that modelling abstraction in terms of clearcut platform-independent and platform-specific qualities is not always applicable [28, 39]. The insight manifests itself in the contributions a progression from MDA to MDE as interaction with industry puts insights from Teaching and Discovery into new contexts. An example of the transition is how the two first publications on Natural Language Generation under Discovery were positioned in an MDA context. The third publication, found under Integration, instead avoids using the terminology of MDA and instead reasons about the relationship between model and text without fixed definitions of abstraction layers.

7 Future Work

For the future we see three main areas that we would like to further pursue – industrial evaluation of natural language generation, promoting life-long learning and agile MDE.

The main concern relating to natural language generation from software models is the lack of industrial evaluation. It is not only in the case of our contributions but in the field as a whole [25]. We aim to initiate a new industrial collaboration to evaluate to which extent practitioners in industry find generated texts useful. As part of such an evaluation the generation procedure will be included to see if and how the technique could be used in general and to determine how much linguistic effort in terms of grammars is needed to fulfill the needs of the readers.

If the role of software and how it is developed keeps changing the coming twenty years as it has the previous twenty years it will be an industrial challenge to ensure

that 'old' employees can keep in touch with new trends. This is already a concern that was mentioned during the interviews at the three companies where the unwillingness to adapt new technologies and paradigms was a common obstacle for systematic change. This opens for new collaborations between industry and academia for continuous learning and emphasises the impact of knowledge transfer.

In our opinion agile MDE warrants more attention. Raising the level of abstraction, automating recurring and complicated tasks as well as full-scale simulations can enable domain experts to become the primary software developers. We still don't know how this scales up in large and distributed software development and if the investment in MDE infrastructure has a positive impact on return of investment in the long run. These are questions we would like to further investigate.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Pearson Education, Inc., Boston, 2007.

[2] S.W. Ambler. *The Elements of UML^{TM}- 2.0 Style*. Cambridge University Press, 2005.

[3] David Ameller. Considering Non-Functional Requirements in Model-Driven Engineering. Master's thesis, Universitat PolitÃÍcnica de Catalunya, June 2009.

[4] Jim Arlow, Wolfgang Emmerich, and John Quinn. Literate Modelling - Capturing Business Knowledge with the UML. In *Selected papers from the First International Workshop on The Unified Modeling Language UML'98: Beyond the Notation*, pages 189–199, London, UK, 1999. Springer-Verlag.

[5] Colin Atkinson and Thomas Kühne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36 – 41, September 2003.

[6] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169 –178, July 2010.

[7] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar – An Embedded Language for Digital Signal Processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Richard L. Baskerville. Investigating Information Systems with Action Research. *Communications of the Association for Information Systems*, 2, 1999.

[9] John Bateman and Michael Zock. Natural Language Generation. In Ruslan Mitkov, editor, *The Oxford Handbook of Computational Linguistics*, Oxford Handbooks in Linguistics, chapter 15. Oxford University Press, 2003.

[10] P Baxter and S Jack. Qualitative Case Study Methodology: Study Design and Implementation for Novice Researchers. *The Qualitative Report*, 13(4):544–559, December 2008.

[11] Mike Beedle, Martine Devos, Yonat Sharon, Ken Schwaber, and Jeff Sutherland. SCRUM: A pattern language for hyperproductive software development. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, pages 637–652. Addison Wesley, 2000.

[12] Grady Booch. *The Unified Modeling Language User Guide*. Pearson Education, 2005.

[13] Ernest L. Boyer. *Scholarship Reconsidered: Priorities of the Professoriate*. Carnegie Foundation for the Advancement of Teaching, Princeton University Press, New Jersey, USA, December 1990.

[14] Ernest L. Boyer. From Scholarship Reconsidered to Scholarship Assessed. *Quest*, 48(2):129–139, 1996.

[15] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, USA, 2012.

[16] A. Brockbank and I. McGill. *Facilitating Reflective Learning In Higher Education*. SRHE and Open University Press Imprint. McGraw-Hill Education, 2007.

[17] Alan W. Brown. Model driven architecture: Principles and practice. *Software and Systems Modeling*, 3(4):314–327, 2004.

[18] John Seely Brown, Allan Collins, and Paul Duguid. Situated cognition and the culture of learning. *Educational Researcher*, 18(1):32–42, Jan-Feb 1989.

[19] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

[20] Håkan Burden. *Three Studies on Model Transformations - Parsing, Generation and Ease of Use*. Licentiate thesis. University of Gothenburg, Gothenburg, Sweden, June 2012.

[21] Håkan Burden and Rogardt Heldal. Natural Language Generation from Class Diagrams. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa 2011, Wellington, New Zealand, October 2011. ACM.

[22] Håkan Burden and Rogardt Heldal. Translating Platform-Independent Code into Natural Language Texts. In *MODELSWARD 2013, 1st International Conference on Model-Driven Engineering and Software Development*, Barcelona, Spain, February 2013.

[23] Håkan Burden, Rogardt Heldal, and Tom Adawi. Pair Lecturing to Model Modelling and Encourage Active Learning. In *Proceedings of ALE 2012, 11th Active Learning in Engineering Workshop*, Copenhagen, Denmark, June 2012.

[24] Håkan Burden, Rogardt Heldal, and Peter Ljunglöf. Enabling Interface Validation through Text Generation. In *In VALID 2013, 5th International Conference on Advances in System Testing and Validation Lifecycle*, Venice, Italy, November 2013.

[25] Håkan Burden, Rogardt Heldal, and Peter Ljunglöf. Opportunities for Agile Documentation Using Natural Language Generation. In *ICSEA 2013, 8th International Conference on Software Engineering Advances*, Venice, Italy, November 2013.

[26] Håkan Burden, Rogardt Heldal, and Martin Lundqvist. Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing. In *6th International Workshop on Multi-Paradigm Modeling MPM'12*, Innsbruck, Austria, October 2012. ACM.

[27] Håkan Burden, Rogardt Heldal, and Toni Siljamäki. Executable and Translatable UML – How Difficult Can it Be? In *APSEC 2011: 18th Asia-Pacific Software Engineering Conference*, Ho Chi Minh City, Vietnam, December 2011.

[28] Håkan Burden, Rogardt Heldal, and Jon Whittle. Comparing and Contrasting Model-Driven Engineering at Three Large Companies. Submitted to ESEM 2014, 8th International Symposium on Empirical Software Engineering and Measurement. To be held in Torino, Italy, September 2014.

[29] Nancy Cartwright. Models: The blueprints for laws. *Philospohy of Science*, 64(1):292–303, December 1997.

[30] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[31] L. Cohen, L. Manion, and K.R.B. Morrison. *Research Methods in Education*. Routledge, 2007.

[32] Andy Crabtree, Peter Tolmie, and Mark Rouncefield. "How Many Bloody Examples Do You Want?" Fieldwork and Generalisation. In Olav W. Bertelsen, Luigina Ciolfi, Maria Antonietta Grasso, and George Angelos Papadopoulos, editors, *ECSCW 2013: Proceedings of the 13th European Conference on Computer Supported Cooperative Work, 21-25 September 2013, Paphos, Cyprus*, pages 1–20. Springer London, 2013.

[33] John Daniels. Modeling with a sense of purpose. *IEEE Software*, 19:8–10, January 2002.

[34] Keith Davis. Methods for studying informal communication. *Journal of Communication*, 28(1):112–116, 1978.

[35] Daniel C. Dennett. Real patterns. *The Journal of Philosophy*, 88(1):27–51, January 1991.

[36] Linda Dickens and Karen Watkins. Action Research: Rethinking Lewin. *Management Learning*, 30(2):127–140, 1999.

[37] E. W. Dijkstra. EWD 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[38] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future. *IEEE Computer*, 42(4):42–52, April 2009.

[39] Ulf Eliasson and Håkan Burden. Extending Agile Practices in Automotive MDE. In *XM 2013, Extreme Modeling Workshop*, Miami, USA, October 2013.

[40] Henry Etzkowitz, Andrew Webster, and Peter Healey. *Capitalizing knowledge: New intersections of industry and academia*. Suny Press, 1998.

[41] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA, 1998.

[42] Donald Firesmith. Modern Requirements Specification. *Journal of Object Technology*, 2(2):53–64, 2003.

[43] Bent Flyvbjerg. Five misunderstandings about case-study research. *Qualitative Inquiry*, pages 219–245, 2006.

[44] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2004.

[45] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[46] Robert B. France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In Lionel C. Briand and Alexander L. Wolf, editors, *International Conference on Software Engineering, ICSE 2007, Track on the Future of Software Engineering, FOSE 2007*, pages 37–54, Minneapolis, MN, USA, May 2007.

[47] Matt Germonprez and Lars Mathiassen. The Role of Conventional Research Methods in Information Systems Action Research. In Bonnie Kaplan, III Truex, DuaneP., David Wastell, A.Trevor Wood-Harper, and JaniceI. DeGross, editors, *Information Systems Research*, volume 143 of *IFIP International Federation for Information Processing*, pages 335–352. Springer US, 2004.

[48] Holger Giese, Stefan Neumann, Oliver Niggemann, and Bernhard Schätz. Model-Based Integration. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, chapter 2, pages 17–54. Springer Berlin/Heidelberg, 2011.

[49] Tony Gorschek, Per Garre, Stig Larsson, and Claes Wohlin. A model for technology transfer in practice. *IEEE Software*, 23(6):88–95, 2006.

[50] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In Giuliano Antoniol, Martin Pinzger, and Elliot J. Chikofsky, editors, *WCRE*, pages 35–44. IEEE Computer Society, 2010.

[51] Richard R. Hake. Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *American Journal of Physics*, 66(1):64–74, 1998.

[52] Cécile Hardebolle and Frédéric Boulanger. Exploring Multi-Paradigm Modeling Techniques. *Simulation*, 85(11-12):688–708, November 2009.

[53] Rogardt Heldal, Daniel Arvidsson, and Fredrik Persson. Modeling Executable Test Actors: Exploratory Study Done in Executable and Translatable UML. In Karl R. P. H. Leung and Pornsiri Muenchaisri, editors, *19th Asia-Pacific Software Engineering Conference*, pages 784–789, Hong Kong, China, December 2012. IEEE.

[54] Rogardt Heldal, Håkan Burden, and Martin Lundqvist. Limits of Model Transformations for Embedded Software. In *35th Annual IEEE Software Engineering Workshop*, Heraklion, Greece, October 2012. IEEE.

[55] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[56] B. Hoffman, H. Mentis, M. Peters, D. Saab, S. Schweitzer, and J. Spielvogel. Exploring Design as a Research Activity. In *Proceedings of the 6th Conference on Designing Interactive Systems*, pages 365–366, University Park, PA, June 2006. ACM, New York, NY.

[57] Pat Hutchings and Lee S. Shulman. The Scholarship of Teaching: New Elaborations, New Developments. *Change: The Magazine of Higher Learning*, 31(5):10–15, September/October 1999.

[58] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven Engineering Practices in Industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 633–642, New York, NY, USA, 2011. ACM.

[59] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, 2014.

[60] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.

[61] Jakob H. Iversen, Lars Mathiassen, and Peter Axel Nielsen. Managing Risk in Software Process Improvement: An Action Research Approach. *MIS Quarterly*, 28(3):395–433, September 2004.

[62] Jean-Marc Jézéquel, Benoît Combemale, Steven Derrien, Clément Guy, and Sanjay Rajopadhye. Bridging the Chasm Between MDE and the World of Compilation. *Journal of Software and Systems Modeling (SoSyM)*, pages 1–17, 2012.

[63] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley-IEEE Computer Society Pr, March 2008.

[64] David Kember and Lyn Gow. Action research as a form of staff development in higher education. *Higher Education*, 23(3):297–310, 1992.

[65] Stuart Kent. Model Driven Engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, 2002. Springer-Verlag.

[66] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28:721–734, August 2002.

[67] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Pearson Education, 2008.

[68] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture^{TM}: Practice and Promise*. Addison-Wesley Professional, Boston, MA, USA, 2005.

[69] Adrian Kuhn, Gail C. Murphy, and C. Albert Thompson. An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *MoDELS*, volume 7590 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2012.

[70] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[71] Kurt Lewin. Action research and minority problems. *Journal of social issues*, 2(4):34–46, 1946.

[72] Jochen Ludewig. Models in software engineering - an introduction. *Software and Systems Modeling*, 2:5–14, 2003.

[73] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[74] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20:14–18, 2003.

[75] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[76] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.

[77] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[78] Farid Meziane, Nikos Athanasakis, and Sophia Ananiadou. Generating Natural Language Specifications from UML Class Diagrams. *Requirements Engineering*, 13(1):1–18, 2008.

[79] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003.

[80] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and MiguelA. Fernandez. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical Software Engineering*, 18(1):89–116, 2013.

[81] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[82] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3):45–77, 2008.

[83] Michael Polanyi. *The Logic of Liberty: Reflections and Rejoinders*. University of Chicago Press, Chicago, Illinois, 1951.

[84] Michael Prince. Does Active Learning Work? A Review of the Research. *Journal of Engineering Education*, 93(3):223–231, 2004.

[85] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML^{TM}*. Cambridge University Press, New York, NY, USA, 2004.

[86] Aarne Ranta. *Implementing Programming Languages - An Introduction to Compilers and Interpreters*, volume 16 of *Texts in Computing*. College Publications, King's College, London, UK, 2012.

[87] Sarah Rastkar, Gail C. Murphy, and Alexander W. J. Bradley. Generating natural language summaries for crosscutting source code concerns. In *27th International Conference on Software Maintenance*, pages 103–112, Williamsburg, VA, USA, September 2011. IEEE.

[88] Ehud Reiter and Robert Dale. *Building Natural Language Generation Systems*. Cambridge University Press, 2000.

[89] Colin Robson. *Real World Research: A Resource for Social Scientists and Practitioner-Researchers*. Regional Surveys of the World Series. Blackwell Publishers, 2002.

[90] Bernhard Rumpe. Agile modeling with the UML. In Martin Wirsing, Alexander Knapp, and Simonetta Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future*, number 2941 in Lecture Notes in Computer Science, pages 297–309. Springer Berlin Heidelberg, January 2004.

[91] P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley, 2012.

[92] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[93] Paulo Sergio Medeiros dos Santos and Guilherme Horta Travassos. Action Research Use in Software Engineering: An Initial Survey. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 414–417, Washington, DC, 2009. IEEE Computer Society.

[94] Walt Scacchi. *Process Models in Software Engineering*, pages 993–1005. John Wiley & Sons, Inc., 2002.

[95] D.C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[96] D.A. Schön. *The Reflective Practitioner: How Professionals Think in Action*. Number TB5126 in Harper Torchbooks. Basic Books Publ., 1983.

[97] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[98] Carolyn B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[99] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, September 2003.

[100] Bran Selic. What will it take? A view on adoption of model-based methods in practice. *Software and Systems Modeling*, pages 1–14, 2012.

[101] Sally Shlaer and Stephen J. Mellor. *Object lifecycles: modeling the world in states*. Yourdon Press, Upper Saddle River, NJ, USA, 1992.

[102] Toni Siljamäki and Staffan Andersson. Performance Benchmarking of real time critical function using BridgePoint xtUML. In *NW-MoDE'08: Nordic Workshop on Model Driven Engineering*, Reykjavik, Iceland, August 2008.

[103] Herbert Alexander Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, MA, 1969.

[104] Ronald A Smith. Formative evaluation and the scholarship of teaching and learning. *New Directions for Teaching and Learning*, 2001(88):51–62, 2001.

[105] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[106] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 101–110, New York, NY, USA, 2011. ACM.

[107] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973.

[108] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons Inc., 2005.

[109] Leon Starr. *Executable UML: How to Build Class Models*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[110] Christine Steeples. Using action-oriented or participatory research methods for research on networked learning. In *Proceedings of the Network Learning Conference*, Lancaster University, England, April 2004.

[111] L Stenhouse. What is action research? University of East Anglia, Norwich, 1979.

[112] J. Stice. *Teaching problem solving*. TA handbook. Texas University, TX, 1996.

[113] Ruth A. Streveler, B.M. Moskal, and Ronald L. Miller. The Center for Engineering Education at the Colorado School of Mines: Using Boyer's Four Types of Scholarship. In *Frontiers in Education Conference, 2001. 31st Annual*, volume 3, pages F4B:11–14, 2001.

[114] Yu Sun, Zekai Demirezen, Marjan Mernik, Jeff Gray, and Barrett Bryant. Is My DSL a Modeling or Programming Language? In *Proceedings of 2nd International Workshop on Domain-Specific Program Development (DSPD)*, Nashville, Tennessee, 2008.

[115] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Preliminary Findings from a Survey on the MD State of the Practice. In *ESEM*, pages 372–375. IEEE, 2011.

[116] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Relevance, benefits, and problems of software modelling and model driven techniques – A survey in the Italian industry. *Journal of Systems and Software*, 86(8):2110 – 2126, 2013.

[117] E. van den Hoven, J. Frens, D. Aliakseyeu, J. Martens, K. Overbeeke, and P. Peters. Design research & tangible interaction. In *TEI'07: Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, pages 109–115, Baton Rouge, LA, February 2007. ACM.

[118] Eelco Visser. A Survey of Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, 2001.

[119] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial Adoption of Model-Driven Engineering – Are the Tools Really the Problem? In Ana Moreira and Bernhard Schaetz, editors, *MODELS 2013, 16th International Conference on Model Driven Engineering Languages and Systems*, Miami, USA, October 2013.

[120] Jon Whittle, Mark Rouncefield, and John Hutchinson. The state of practice in model-driven engineering. *IEEE Software*, 2013.

[121] Robert K. Yin. *Case Study Research: Design and Methods*. SAGE Publications, California, fourth edition, 2009.

Appendix A:
Scholarship of Teaching

Paper 1:
How Difficult Can it Be?

Published as:
Executable and Translatable UML – How Difficult Can it Be?
Håkan Burden[1], Rogardt Heldal[1] and Toni Siljamäki[2]
[1] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
[2] Ericsson AB
APSEC 2011
18th Asia Pacific Software Engineering Conference
Ho Chi Minh City, Vietnam
5-8 December 2011

1 Introduction

In Model-Driven Architecture (MDA; [25]), the requirements and responsibilities of the system are given a structure by the use of software models in a Computationally-Independent Model, the CIM, often referred to as the domain model [22]. Features such as specific algorithms and system architecture are defined by the next layer of models, the Platform-Independent Models, the PIM. The PIM has no ties towards the hardware nor the programming languages that will in the end realise the system. Such information is added to the Platform-Specific Model, the PSM. As a result the software models of the CIM and the PIM can describe many different implementations of the same system. The models become reusable assets [20] serve both as a description of the problem domain and a specification for the implementation, bridging the gap between problem and solution.

Executable and Translatable UML (xtUML; [19, 33]) is an extension of UML [24] with models that can be executed and translated into code through model compilers. In MDA terms, the xtUML model is an executable PIM that can be automatically transformed into a PSM. The efficient and consistent transformation from a PIM specified using xtUML to a PSM has been tested and proven in previous work [31, 6]. But it is still an open question how much expertise that is required to use xtUML as an executable modelling language for PIMs.

1.1 Motivation

For ten years we have given a university course where teams of students go through the different tasks of an MDA process; from analysis to implementation by designing the system using UML models. The process is illustrated in Figure5. The numbers for each activity in the process are specific to the course and state the maximum number of hours for each student.

The analysis phase was used to capture the business rules of the problem domain in models that satisfy the requirements of the system. The focus during the analysis phase was thus on understanding the problem domain by using activity diagrams, use cases and conceptual class diagrams [15, 24].

The second phase of the process, the design phase, was where more detailed UML diagrams were used such as interaction, state chart, class and component diagrams. Even though this phase should be important for the overall process, we found that it contributed little to the overall system for most teams (in Lean terms the models represent waste [27]). The diagrams were incomplete, lacking necessary details in structure and behaviour. The only way of testing the models was through model inspection, making it a matter of opinion when the models are complete [28]. Another problem with the models was that they were inconsistent with each other leading to complications about which model to follow in the transformation to source code. The impact of the problems vary depending on how important the models are in the development process and when and how the inconsistencies are shown [12, 14, 16, 34].

The design phase was followed by an implementation phase were the students manually transformed their models into Java. This meant that it took months before the students could test their analysis and design. This is a problem shared with

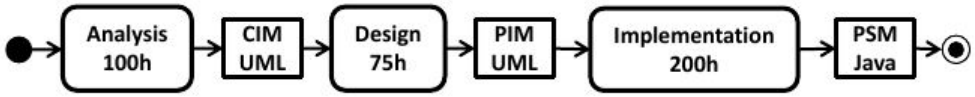

Figure 5: The old software development process

industry [18].

Since the code is manually written with the models as a guide it also means that there is a difference in the interpretation of the problem between the UML model and the hand-written code. By default you reanalyze the problem when you start writing the code, and you often come up with a different solution compared to the modelled solution. Eventually the model and the hand-written code diverge, so the only way to really understand what's going on in the system is to study the hand-written code. The model may then serve as a quick, introductory overview of the system, but it may also be incorrect as soon as you stop updating the model for reflecting the changes made to the hand-written code. This notion of architecture erosion is a well-known problem and is still being reported on [4, 17, 26].

1.2 Aim and Research Question

As a result from collaboration between industry and academia we came up with the idea to use xtUML to model the component- and class diagrams and the statemachines instead of UML in the design phase. If the change of modelling language is successful we will get rid of the inconsistency problems. With an executable PIM it should be possible for the students to test and validate their design decisions without having to implement them in Java first. And when the models behave as expected and the design phase is complete, all functionality of the system should be captured in the models [28, 33]. In the long run this will mean that a lot of the work that was done in the implementation phase can be replaced by generating the code straight from the models, leading to shortened implementation times and consistency between models and code.

Swapping UML for xtUML is not a one-to-one substitution. If it is a matter of opinion when a UML model is complete, an xtUML model is complete when all test cases return the expected results [28]. So, xtUML is more than the graphical syntax, the models have to be given semantics to be executable. In addition, test cases have to be modelled, executed and evaluated. These additions demand that the xtUML tool is more than a drawing tool.

Will the immediate and constant feedback that is given from executing the test cases compensate for the increase in modelling effort? Or will the added effort for learning xtUML take so much time that there is no left for modelling? This concern is re-phrased into our research question:

"Can teams of four novice software modellers solve a problem that is complex enough to require the full potential of xtUML as a modelling language within a total of 300 hours?"

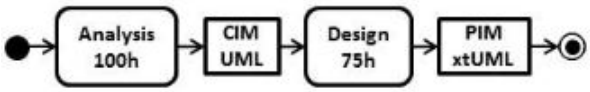

Figure 6: The proposed software development process

To answer our research question we scrapped our old course in favour of a new one that follows the process seen in Figure 6. Instead of spending 200 hours implementing the design to be able to test and verify it, testing will now be a part of the design phase. Just as for the process in Figure 5 the number of hours in each step states the maximum for each student. The introduction of the new design phase was done as a case study with the ambition to explore and explain the transition and its implications.

1.3 Contribution

Earlier contributions has shown how Executable and Translatable UML enables MDA [19, 20, 28], the reusability of the PIM has been reported on in [3] and the efficiency of the transformations from PIM to PSM is illustrated by [31]. Our contribution shows that xtUML as a technology is mature enough to to be used by novices to design executable PIMs.

1.4 Overview

In the next section we go into more detail of xtUML and how it can be used. In section III we explain how our subjects made use of xtUML to develop and test a hotel reservation system. Our findings and their validity are presented in section IV. In section V we discuss the implications of our results and in section VI we relate our own case study to previous work. This is followed by a conclusion and some ideas about further investigations regarding the usage of xtUML.

2 Executable and Translatable UML

The Executable and Translatable Unified Modeling Language (xtUML; [19, 28, 33]) evolved from merging the Shlaer-Mellor method [30] with the Unified Modeling Language (UML, [24]).

2.1 The Structure of xtUML

Three kinds of diagrams are used for the graphical modeling together with a textual action language. The diagrams are component diagrams, class diagrams and state-machines. There is a clear hierarchical structure between the different diagrams; state-machines are only found within classes, classes are only found within components. The different diagrams will be further explained below, together with fragmentary examples taken from the problem domain given to the students, a hotel reservation system.

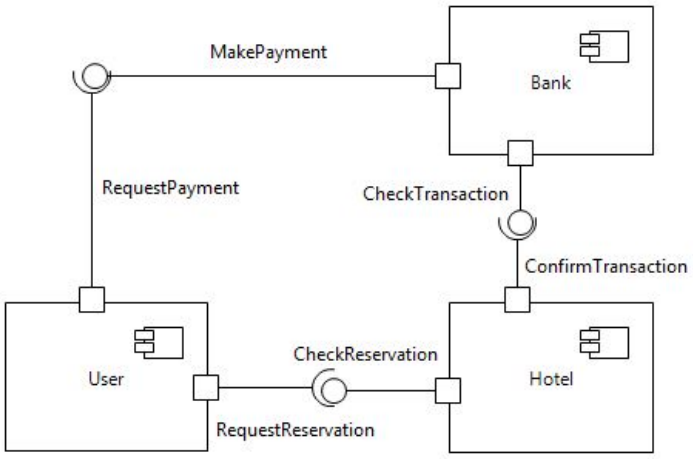

Figure 7: An xtUML component diagram

2.1.1 Component Diagrams

The xtUML component diagram follows the definition given by UML. An example of a component diagram can be found in Figure 7. In this diagram the hotel domain depends on the bank for checking that a transaction has gone through as part of the process of making reservations. The User component represents a users of the system and this is where the test cases are placed.

2.1.2 Class Diagrams

In Figure 8 we have an example of an xtUML class diagram. It describes how some of the classes found in the Hotel component relate to each other. I.e. a Room can be related to any number of Reservations (shown by an asterisk, *) but a Reservation has to be related to at least one Room (visualized by 1..*).

The xtUML classes and associations are more restricted than in UML. We will only mention those differences that are interesting for our case study. A feature such as visibility constraints on operations and attributes does not exist. They are therefore accessible from anywhere within the same component. In UML the associations between classes can be given a descriptive association name while in xtUML the association names are automatically given names on the form RN where N is a unique natural number, e.g. Room is associated to Reservation over the association R2.

2.1.3 State Machines

In the class diagram in Figure 8 the Reservation class has both an instance and a class state machine which is indicated by the small figure in the top-left corner of the class. The instance state machine can be found in Figure 9. This state machine covers the first four states of the Reservation procedure, e.g. from the second state, Get rooms,

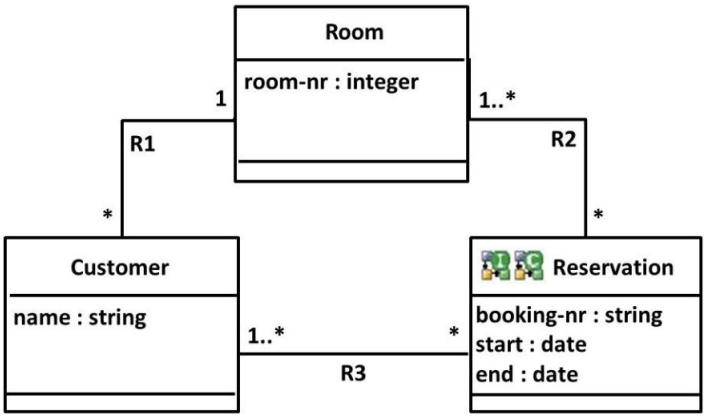

Figure 8: An xtUML class-diagram

it is possible to reach the third state, Lock rooms, by requesting the rooms. If there are no available rooms you return to the initial state were you can start a new search. Each instance of Reservation has its own instance statemachine that starts running when the Reservation is created.

A class-based state-machine is shared among all instances of a class and starts running as soon as the system starts, like a static process. For shared resources, such as rooms, a class state-machine can be used to ensure that only one reservation instance can book a room at any time.

2.1.4 Action Language

An important difference between standard UML and xtUML is that the latter has a textual programming language that is integrated with the graphical models, sharing the same meta-model [7, 30].

The number of syntactical constructs is deliberately kept small. The reason is that each construction in the Action Language shall be easy to translate to any programming language (such as Java, C or Erlang) enabling the PIM to be reused for different PSMs [3].

There are certain places in the models were Action Language can be inserted, such as in operations, events and states. Over the years a number of different Action Language have been implemented [19] and in 2010 OMG released there own standard [23].

2.2 Interpretation and Code Generation

Since xtUML models have unambiguous semantics all validation can be performed straight on the xtUML model by an interpreter. During the execution of the test cases an object model is created. The object model includes all class instances with their current attribute values and by which associations they are linked to each other.

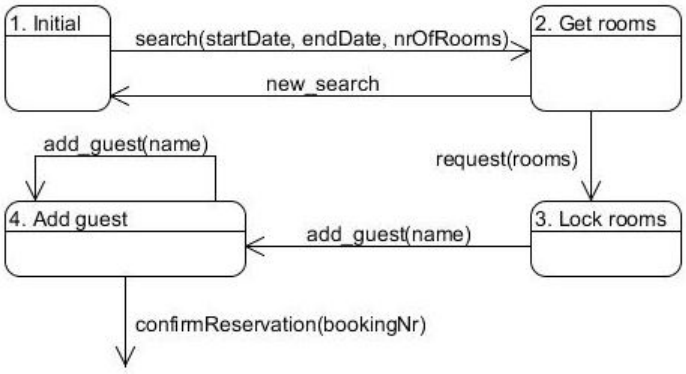

Figure 9: A partial xtUML statemachine

During execution all changes of the association instances, attribute values and class instance are shown [15] as well as the change of state for classes with statemachines in the object model.

The xtUML models can be translated into Platform-Specific Models by model compilers. Since the Platform-Specific code is generated from the model, it is possible for the code and the models to always be in synchronization with each other since all updates and changes to the system are done at the PIM-level, never by touching the code. The efficiency of the generated code has been reported on by [31]. Ciccozzi et al. have used the model compiler to generate test cases for the PSM [6].

3 Case Study Design

To answer our research question:

"Can teams of four novice software modellers solve a problem that is complex enough to require the full potential of xtUML as a modelling language within a total of 300 hours?"

we have both in 2009 and 2010 let our students use xtUML to design hotel reservation systems. The resulting models have been inspected and compared against our evaluation criteria.

3.1 Subject and Case Selection

3.1.1 The Subjects

Our subjects were students in the final year of their bachelor programs in computer science and software engineering. Their prior knowledge of modelling is limited to class diagrams but they are used to programming in an object-oriented paradigm using Java. In our curriculum the students do two courses in parallel with a working week of 50 hours, so we expect the subjects to work 25 hours a week on our course. A team of four subjects is expected to do a total of 100 hours per week.

3.1.2 The Case

We chose a domain that the subjects could relate to and have some prior knowledge about. The idea is that the subjects shall focus on modelling, not learning a new subject matter. The domain should also have distinct concepts so that an object-oriented solution made sense and have problems where it is natural to use state machines. We also wanted the domain to include problems with algorithmic complexity. Our last requirement was that the domain should represent an open-ended problem so that there is not one right solution. A system for handling hotel reservations seemed to fit all our requirements.

In the hotel domain reservations, customers and rooms are all examples of distinct concepts. The booking process itself has a chain of states that it is natural to control with a statechart, while finding all the possible matches to a set of search criterias for a reservation is an algorithmic problem. These two together, controlling the order of events and searching for rooms, meant that the domain poses the problem of access and allocation of shared and limited resources.

The new design phase was given three weeks, just as the previous design phase. The work was done in teams of four subjects, with a total workload of 75 hours per subject.

The subjects used BridgePoint [5] from Mentor Graphics [21] to design the xtUML models. There were three 90-minute lectures related to xtUML and BridgePoint. Two of these lectures were given by industry representatives, one from the tool vendor Mentor Graphics and one from Ericsson AB as users of BridgePoint. The subjects were encouraged to study xtUML on their own and we recommended them to read about xtUML [19] and clear design [32]. Each team had half an hour a week with a researcher to discuss design issues. Besides lecturing and supervising on design issues the researchers played the role of project owners.

In 2010 we added limited tool support for the subjects. A subject from 2009 used a total of 22 hours spread over the three weeks to help the subjects of 2010 with BridgePoint. This meant that each team had access to less than an hour of tool training for the entire design phase.

3.2 Data Collection Procedures

We have done two data collections, in 2009 and 2010 respectively. In 2009 there were 88 subjects split into 22 teams. In 2010 we had 108 subjects divided into 28 teams, with four (sometimes three) members per team. We used three forms of data collection; model evaluations, informal discussions and a questionnaire. Model evaluations and informal discussions were used both times but the questionnaire was only used in 2010.

3.2.1 Evaluating the xtUML Models

The evaluation of the xtUML models was done immediately after the design phase and took a whole week to complete. Each team was given 20 minutes to demonstrate their system and to run their tests. Thereafter there was 20 minutes to discuss issues related to their models. Every model was evaluated by two researchers against the evaluation criteria that are specified below. The subjects of each team were present throughout

the evaluation, permitting a discussion on the how the criteria on functionality and structure had been interpreted and implemented in the model. A short description of each model with our comments was taken down in a spread sheet.

3.2.2 Informal Discussions

We thought it vital to have an informal opportunity for the subjects to discuss their experiences throughout the design phase. This was an important opportunity for us to get more in-depth information into the problems and discoveries that the subjects had encountered when using BridgePoint to model xtUML. Since we did not know what to expect for outcome in 2009 we wanted the subjects to have the opportunity to drop us an e-mail, come by our offices or use the lectures for addressing those issues they found urgent. This proved to be a valuable source for data collection, so valuable that we kept it in 2010. The drawback is that it is not a procedure that is always possible to document or systemize.

3.2.3 Questionnaire

One of the most important things that became evident from the informal discussions in 2009 was that the subjects found the learning threshold stressing under the time constraint. Besides introducing tool support in 2010 to ease the subjects' stress we conducted a questionnaire. The aim of the questionnaire was to get a better view of how much time the subjects spent on getting confident in using BridgePoint.

3.3 Evaluation Criteria

Before the subjects started to develop their models we gave them evaluation criteria. The reason was to have a clear idea for both the researchers and subjects of what we expected from the teams.

Based on the use cases from the analysis phase the subjects should come up with executable tests. By running the tests it should be possible to validate that the system is behaving as specified by the CIM. This meant that the object model had to show all relevant changes for objects, associations, attributes [15] and states after a test had been run. At least one test case should be in conflict with the business rules of the system.

However, this does not guarantee that the models are well-structured nor readable [13, 28, 32]. Therefore the criteria enforced an object-oriented design.

For the class diagram we did not accept models with a central object representing the system [32]. Due to the lack of visibility constraints in xtUML we stated that the only way to obtain or change the value of an attribute should be through operations. It should only be possible for a class instance to call another class instance if they are linked by associations. This is so that the dependencies between the class instances are explicit in the class diagram. We wanted all the meaningful associations and concepts in the class diagram. We requested names on classes, attributes, operations and variables that were relevant for the domain.

For the state-machines it was necessary to include all the states and transitions relevant for capturing the lifecycle of the class where it resides. The name of events and states should be meaningful. To ensure that the subjects made use of the power of state machines we required that they should be used for modelling the reservation process.

4 Results

Over the two years that we have used the new design phase we have evaluated 50 xtUML models. Of these 43 have fulfilled the success criteria. The subjects had to overcome a learning threshold before they got confident in using BridgePoint to develop their xtUML models. All in all, we can answer our research question by stating that the teams did manage to use the full power of xtUML within 300 hours.

4.1 Results from Evaluating the Models

In 2009 all 22 teams came up with an executable model, capturing at least the minimum functionality. 18 of the 22 teams, equivalent to 83% managed to come up with a model within the time frame that met our criteria for a successful model. The models that did not meet the criteria had either a monolithic class that represented the whole system and/or not used the associations to access class instances. Most teams did not use components, but that was not a strict criteria either. This was a shortcoming of the criteria as we had wanted to see the subjects use components, since it would have added a whole new level of abstraction to their models. On the other hand, we were encouraged by the fact that all teams had delivered testable models that fulfilled the criteria for functionality.

Three teams came up with models that went beyond our expectations. They had used components and modelled more functionality than we thought possible. One of the three teams had even made extensive use of design patterns [10].

In 2010 there were 28 teams. 25 of these managed to deliver executable models within the time frame. This is an increase from 82% to 89%, compared to 2009. One of the teams that failed to specify an executable model did so due to unresolvable personal issues within the team. The other two teams misjudged how long time it would take to come up with a model and were not finished in time due to their late start. In 2009 all teams succeeded to come up with executable models compared to 89% in 2010. But this time all teams had components and interfaces with the consequence of using the full power of xtUML. This was not an intended outcome of the added tool training but highly appreciated even so!

4.2 Outcomes From the Informal Discussions

The subjects are used to programming in Java. In 2009 it took the subjects some time before they started to reason about objects and programming in an xtUML-way. When they encountered a problem many of them expected a solution using detailed

Java-specific datastrucutres or libraries. In contrast BridgePoint is mostly used for embedded systems were the companies have their own private libraries.

Particularly state-machines were difficult to use since they had no counterpart in Java. It is not possible for us to say if this is due to that Java is the only language they are used to or if it is due to the more abstract level of reasoning in xtUML.

BridgePoint is a powerful tool; enabling modelling, execution of models and translation to source code. All the functionality makes it a complex tool. BridgePoint is also to a large extent menu-driven — many of the design choices are implemented by choosing from drop-down menus and tool panels. The challenge is to get used to all the different combinations of choices that are needed for elaborating the design. The version we used is a plug-in for Eclipse which in itself has a number of features to get used to.

In reaction to the problems concerned with BridgePoint as a tool we decided to use one of the subjects from 2009 for tool training in 2010. The intention was that this would mean less time spent on understanding the tool and more time to spend on developing the models.

In 2010 the subjects requested a version control system so that they could more easily split the design work between themselves. This was never an issue in 2009 and a sign of more confident subjects. If this is a consequence of the tool training or not is to early to answer.

4.3 Experienced Learning Threshold

In 2010 we used a questionnaire to get a better idea of how the subjects experienced BridgePoint. The question we asked was "How many hours did you spend learning BridgePoint before you got confident in using the tool?"

In total 90 of 108 subjects answered the question. Besides the answers given in Figure 10 six subjects answered that they never became confident and one subject had no comment. The number of hours it took to become confident are given on the x-axis, the number of subjects for a given number of hours is displayed along the y-axis. This means that 27 subjects answered that it took 30 hours to become confident in using BridgePoint.

In Figure 11 the x-axis carries the same information as in Figure 10 while the y-axis now displays the total number of confident subjects for a given time, e.g. after 30 hours a total of 57 subjects felt confident in using BridgePoint. After 40 hours this figure had risen to 75 subjects.

4.4 Relevance to Industry

We wanted to solve our consistency problems by using xtUML instead of UML. In our view this was successful. But we believe that xtUML can have a larger impact than just solving the problems we had with our old development process 5. From the outcomes of the the evaluation of the models, the subjects' own figures for the confidence threshold for using xtUML and the informal discussions we propose a hypothesis.

We state that xtUML, both as a modelling language and tool, is easy to learn and use without any prior experience of software modelling. It is enough to have the

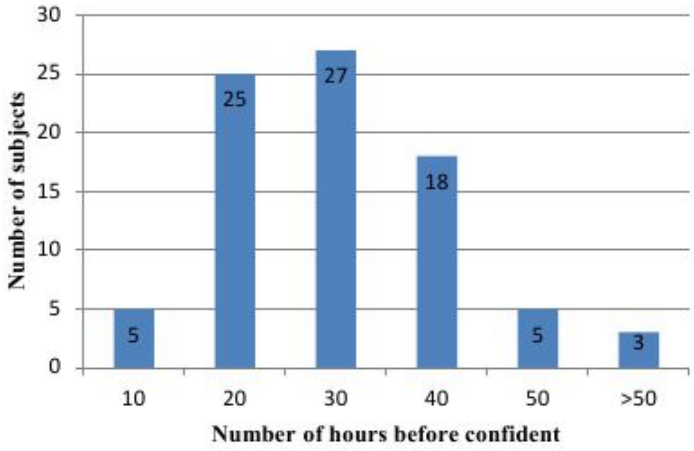

Figure 10: Number of hours that the subjects needed to become confident in using BridgePoint

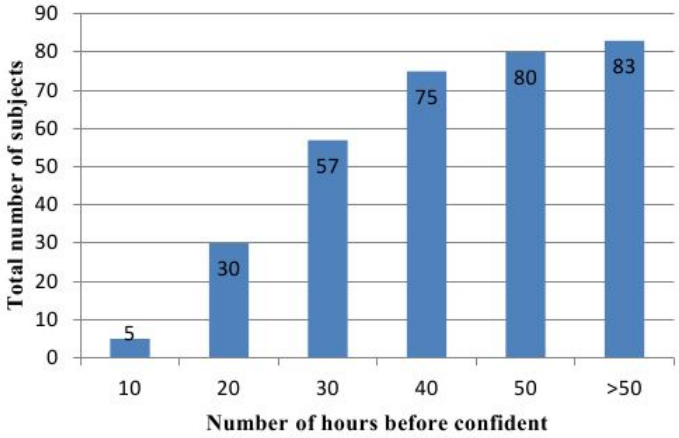

Figure 11: Total number of subjects that are confident in using BridgePoint depending on time

programming experience equivalent to a bachelor student in computer science. Our subjects managed to learn and understand the full expressivity of xtUML within 75 hours, and that includes using asynchronous events in statemachines and sending and receiving signals between components as well as designing the models to an appropriate level of detail. If it is possible for us to manage with the transition from UML to xtUML it should be possible to do so in industry as well.

4.5 Evaluation of Validity

We have analysed our results using the classification of validity as defined by Wohlin et al. [35], Yin [36] and Runeson and Höst [29].

4.5.1 Construct Validity

Our solution to the problems we had earlier is set in the same context as the problem was. Our subjects have the same background and experience as previous students, just as before the subjects have the necessary domain knowledge from the analysis phase, they work under the same time constraints and come up with the same kind of models. The only thing that has changed is the modelling language, which is what we want to evaluate.

The subjects were present during the model evaluation. This was done in order to reduce our bias in interpreting their work and how it related to our validity criteria.

In order to assess the quality of the software models we have specified evaluation criteria. In 2009 only three of 22 teams defined two or more components and the interfaces between them. Since the 28 teams in 2010 also had models with several components with defined interfaces we can see that the subjects managed to use the full power of xtUML.

The figures for the number of hours to overcome the learning threshold of BridgePoint were estimations done by the subjects themselves. There might be variations among the subjects of the definition of when the threshold is passed. Our experiences from Ericsson and our own observations correlate with the estimations of the subjects. However, there is a possibility that some subjects have exaggerated or under-estimated their figures due to social factors (reactions towards the tool, researchers or team members). The exact nature of the learning threshold for xtUML will be dependent on the background of the subjects and which tool that is used.

4.5.2 Internal Validity

The evaluated models are developed on the basis of the CIM from the analysis phase, so that everything the subjects need to know about hotel reservation systems should be found in their CIM. This means that they should not need to spend time during the design phase on anything else than learning and using xtUML.

Even if the evaluation criteria have influenced the nature of the results, by defining what we expected from the subjects' models, the process of getting there was by using xtUML.

In the first run of the experiment we did not know what to expect for results. In the second run we had expectations based on the results from the first run. Therefore we were careful to make sure that we as researchers had the same roles towards the subjects in both runs, which led us to let a subject from 2009 take care of the tool training in 2010. We still cannot neglect that our changed expectations might have influenced the outcome in 2010, even if we did not get the results we were expecting. On the other hand this is always the case in situations where you want to replicate research that involves human beings. This is also a threat to the reliability of our results.

It is possible that the subjects have been sharing insights and experiences throughout the case study. We knew this could be the case from the start and that was one reason why we wanted an open-ended problem. During the evaluations we have seen 50 unique models which implies that all teams have had their own process to come up with the models.

4.5.3 External Validity

Today's students are tomorrow's employees. If our subjects can master xtUML it should also be possible for software developers in industry to do so. This is also claimed by Kitchenham et al. [11] who state that students can be used in research as long as it is the evaluation of the use of a technique by novice users that is intended. We can expect software developers in industry to have at least the same competence as our subjects.

The size of the problem given to the subjects is smaller than most indutrial sized tasks. Our domain has a certain level of complexity and was chosen from our collaboration between industry and academia. By handling the access and allocation of the shared resources within the hotel domain, we made sure that the subjects had to solve a non-trivial task.

4.5.4 Reliability

Since the evaluation of the models is subjective there was at least two researchers present at every evaluation in order to reduce the risk of bias and inconsistencies between evaluators. We also used the criteria for a successful model to ensure that the evaluation is less subjective, making the results less dependant on a specific researcher.

BridgePoint was chosen by the authors based on the fact that a team within Ericsson think that this is one of the best MDA solutions today. After we had made our decision on which tool we would prefer to use we contacted Mentor Graphics in order to start a collaboration with them. Mentor Graphics never influenced us on which tool to use. Using another xtUML tool might give other results, especially regarding the threshold, both in figures and what is seen as problematic.

5 Discussion

In our previous MDA process, given in Figure 5, our students manually transformed the PIM into a PSM. From our previous experiences of using xtUML as a code generator [31, 6] we know that this manual transformation can be automated. However, to raise the quality of the generated code the PIM needs to be manually enhanced by a marking model [1, 20, 22, 31]. The generated code will then be sufficient for an embedded system. For systems that interact with human users it will also be necessary to develop the needed user interfaces. All in all the introduction of xtUML should enable a less time-consuming implementation phase compared to our old MDA process.

6 Related Work

There is a previous experience from using xtUML in the context of computing education reported in [8, 9]. One of their motivations for using xtUML in a modelling course is that they found UML to large, ambiguous and complex. In contrast, xtUML models are unambiguous and easier to understand than UML models. The possibility of verifying the models to see if they meet the requirements is important in order to give the students feedback on their modelling. The authors have used xtUML both for specifying a web application and for 3D drawing software.

By using xtUML in a similar context as ours their work strengthens our claim that xtUML can be used by novice software modellers. However, there work does not report any results from letting undergraduate students use xtUML; there are no clear criteria for what was seen as a successful project and subsequently no reports on how many students that managed to complete the task. It is also unclear how much time the students spend on their models. Another important difference is that we find UML useful for defining the CIM.

Both [19] and [28] describe xtUML in the context of MDA. Starting of from use cases they develop PIMs by using xtUML. While the main focus is on developing an executable PIM both books takes the reader from CIM to PSM. The main differences lie in the choice of tool and in how they choose to describe and explain xtUML.

We have made extensive use of both books as a source of inspiration for how to work with xtUML and as recommended literature for the subjects for obtaining executable PIMs in an MDA context. These are the most cited books on how to use xtUML and they are detailed in how this is accomplished. However, they do not mention the effort for learning xtUML nor the level of expertise needed to use xtUML as a modelling language for PIMs.

7 Conclusions and Future Work

7.1 Summary

Even if the subjects spend the first week of the design phase in order to learn BridgePoint the fact that the models have a testable behaviour more than compensates for the increase in effort. Our subjects used the test cases to refine their models until they met the criteria. As a consequence their PIMs had the necessary detail and structure as defined by the CIM. This was possible since xtUML gave them constant and immediate feedback on all their design decisions.

In contrast, UML models are not executable. More or less the only way of checking the quality of a UML model is by performing a model review. This is a powerful method for improving on UML models but it is also time consuming. And it can be hard to catch the mistakes in complex systems. It becomes a matter of opinion when a model can be considered complete. In contrast an xtUML model is complete when it only delivers expected output for all test cases.

Previous work has shown that xtUML enables MDA by the reusability of the PIMs, the efficient transformation from PIM to PSM and by solving the problems of

inconsistencies within the PIM. Our work shows with what little effort and expertise it is possible to develop PIMs, using the full expressivity of xtUML. This implies that Executable and Translatable UML is a technology that is ready to be used within industry.

7.2 Future Work

We are looking at the possibilities to expand the new course so that it covers the entire MDA-process, from CIM to PSM. Issues we want to investigate is how difficult it is to mark the PIMs for an efficient transformation to PSMs, the effort for deploying the generated code on a platform with the required user interfaces and how much time we can save compared to the old development process, illustrated in Figure 5.

In addition, we want to investigate the reasons behind the socio-technical gap [2] to understand why xtUML is not used more within industry and software development.

Acknowledgment

The authors would like to thank Staffan Kjellberg at Mentor Graphics; Stephen Mellor; Leon Starr at Model Integration; Dag Sjøberg at University of Oslo; Jonas Magazinius, Daniel Arvidsson, Robert Feldt and Carl-Magnus Olsson at Computer Science and Engineering in Gothenburg.

Bibliography

[1]

[2] Mark S. Ackerman. The intellectual challenge of cscw: The gap between social requirements and technical feasibility. *Human-Computer Interaction*, 15:179–203, 2000.

[3] Staffan Andersson and Toni Siljamäki. Proof of Concept - Reuse of PIM, Experience Report. In *SPLST'09 & NW-MODE'09: Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, Tampere, Finland, August 2009.

[4] Jan Bosch. Architecture in the age of compositionality. In Muhammad Babar and Ian Gorton, editors, *Software Architecture*, volume 6285 of *Lecture Notes in Computer Science*, pages 1–4. Springer Berlin, Heidelberg, 2010.

[5] BridgePoint. http://www.mentor.com/products/. Accessed 13th January 2012.

[6] Federico Ciccozzi, Antonio Cicchetti, Toni Siljamäki, and Jenis Kavadiya. Automating test cases generation: From xtUML system models to QML test models. In *MOMPES: Model-based Methodologies for Pervasive and Embedded Software*, Antwerpen, Belgium, September 2010.

[7] Michelle L. Crane and Jürgen Dingel. Towards a Formal Account of a Foundational Subset for Executable UML Models. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS, 11th International Conference on Model-Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 675–689, Toulouse, France, September 2008. Springer.

[8] S Flint and C Boughton. Executable/Translatable UML and Systems Engineering. In Alan McLucas, editor, *Systems Engineering and Test Evaluation Conference (SETE 2003)*, Canberra, Australia, 2003.

[9] Shayne Flint, Henry Gardner, and Clive Boughton. Executable/Translatable UML in computing education. In *ACE'04: Proceedings of the sixth conference on Australasian computing education*, pages 69–75, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[11] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28:721–734, August 2002.

[12] Christian F. J. Lange. Improving the quality of UML models in practice. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 993–996. ACM, 2006.

[13] Christian F. J. Lange, Bart Du Bois, Michel R. V. Chaudron, and Serge Demeyer. An experimental investigation of UML modeling conventions. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2006.

[14] Christian F. J. Lange and Michel R. V. Chaudron. Effects of defects in UML models: an experimental investigation. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 401–411, New York, NY, USA, 2006. ACM.

[15] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[16] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009.

[17] N. Mellegård and M. Staron. Methodology for Requirements Engineering in Model-Based Projects for Reactive Automotive Software. In *European Conference on Object-oriented Programming (ECOOP)*, Paphos, Cyprus, 2008.

[18] Niklas Mellegård and Miroslaw Staron. Characterizing model usage in embedded software engineering: a case study. In Ian Gorton, Carlos E. Cuesta, and Muhammad Ali Babar, editors, *ECSA Companion Volume*, ACM International Conference Proceeding Series, pages 245–252. ACM, 2010.

[19] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[20] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[21] Mentor Graphics. http://www.mentor.com/. Accessed 13th January 2012.

[22] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003.

[23] OMG. Concrete Syntax for UML Action Language (Action Language for Foundational UML - ALF). http://www.omg.org/spec/ALF/. Accessed 30th April 2011.

[24] OMG. OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3. http://www.omg.org/spec/UML/2.3/. Accessed 11th September 2010.

[25] OMG. MDA. http://www.omg.org/mda/, Accessed January 2011.

[26] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17:40–52, October 1992.

[27] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[28] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML^{TM}*. Cambridge University Press, New York, NY, USA, 2004.

[29] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[30] Sally Shlaer and Stephen J. Mellor. *Object lifecycles: modeling the world in states*. Yourdon Press, Upper Saddle River, NJ, USA, 1992.

[31] Toni Siljamäki and Staffan Andersson. Performance Benchmarking of real time critical function using BridgePoint xtUML. In *NW-MoDE'08: Nordic Workshop on Model Driven Engineering*, Reykjavik, Iceland, August 2008.

[32] Leon Starr. How to Build Articulate UML Class Models. http://knol.google.com/k/leon-starr/how-to-build-articulate-uml-class-models/2hnjef6cmm97l/4. Accessed 24th November 2009.

[33] Leon Starr. *Executable UML: How to Build Class Models*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[34] Ragnhild Van Der Straeten. Description of UML Model Inconsistencies. Technical report, Software Languages Lab, Vrije Universiteit Brussel, 2011.

[35] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers Norwell, MA, USA, 2000.

[36] Robert K. Yin. *Case Study Research: Design and Methods*. SAGE Publications, California, fourth edition, 2009.

Paper 2:
Pair Lecturing to Model Modelling

Published as:
Pair Lecturing to Model Modelling and Encourage Active Learning
Håkan Burden[1], Rogardt Heldal[1] and Tom Adawi[2]
[1] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
[2] Engineering Education Research
Chalmers University of Technology
ALE 2012
11th Active Learning in Engineering Education Workshop
Copenhagen, Denmark
20-22 June 2012

1 Introduction

In this paper, we describe an ongoing action research project [13] to improve teaching and learning in the course Model-Driven Software Development [6, 7] given by the Department of Computer Science and Engineering. The majority of the students are third year students and after taking the course the students should be better able to analyze and specify software through models. The course is based on lectures and a team project. Over the years, we have however noticed that students have difficulties in applying the theory presented in the lectures to their projects. We therefore felt that the lectures could be improved to better prepare the students for tackling the project, and hence to bridge the gap between theory and practice. The two research questions we have decided to address in this paper are:

1. Can pair lecturing encourage students to take a deep approach to learning in lectures?

2. What are the pros and cons of pair lecturing for students and teachers?

A deep approach to learning involves *"the critical analysis of new ideas, linking them to already known concepts and principles, and leads to understanding and long-term retention of concepts so that they can be used for problem solving in unfamiliar contexts"* [12].

Stice [20] argues that the main reason why students have trouble applying what they know to novel situations is that the teacher seldom models the entire problem solving process, including assumptions, alternative strategies and evaluation of results. What the students get to see is often a neat solution - the product but seldom the process behind the product. Because the students *"own attempts to solve problems seem more painful and often are unsuccessful, they may think that the professor is a genius (or a magician), or that they are dumb. Odds are that neither is the case"* [20]. Bain [1] studied, over a period of fifteen years, the teaching methods of nearly one hundred successful college teachers from a variety of disciplines and concludes that: *"The most effective teachers use class time to help students think about information and ideas the way scholars in the discipline do. They think about their own thinking and make students explicitly aware of that process, constantly prodding them to do the same"*. Or in the words of Buckley [5] the teachers *"model the competence they try to impart, forming the students by their example of interaction as much as by their words."*

There is also a large body of research demonstrating that students learn more when they are actively involved during lectures [10, 18]. Felder [9] defines active learning as *"anything course-related that all students in a class session are called upon to do other than simply watching, listening and taking notes"*. Examples of well-known active learning exercises include the pause procedure, the one-minute paper, and the think-pair-share activity [16]. Bonwell and Eison [3], who popularized this approach to teaching, emphasize that *"to be actively involved, students must engage in such higher-order thinking tasks as analysis, synthesis, and evaluation"*. When lectures are delivered using PowerPoint slides as the main presentation medium it is easy to fall into the trap of covering too much material and moving too quickly through the material.

And as Biggs [2] has pointed out, *"coverage is the greatest enemy of understanding"*. Moreover, when slides serve as a fixed and linear script for the lecture, it can be difficult to adjust the flow of the lecture to the student interaction. As a consequence, there is a risk that students become passive consumers of information.

2 Methodology and Method

In this study, we have therefore drawn on cognitive apprenticeship [4] as a theoretical framework for teaching and learning. A key component of cognitive apprenticeship is that the teacher models and verbalizes the cognitive processes that experts engage in as they solve problems. This act of making thinking visible should be carried out in collaboration with students and by using real-life examples.

We wanted the software models to take form in the lecture hall in interaction with the students so that the assumptions and motivations for alternative solutions became explicit. But this immediately raised four challenges:

1. Getting students to actively participate in the lectures;
2. Finding examples from real life where the students have sufficient background knowledge;
3. Being able to correctly interpret comments and questions from the students;
4. Being able to cover the necessary material so the objectives of the lectures are fulfilled;

We believed that we would be better able to deal with these challenges through pair lecturing. One of us was responsible for preparing the lectures and another one carried out the drawing of the model on the whiteboard during the lectures. In this way it was natural for the one drawing the model to question and discuss with the first lecturer what the purpose and necessary level of detail of the model was, as well as the pros and cons of alternative solutions. This discussion between the lecturers could motivate students to actively engage with the material and perhaps also participate in the lecture. Moreover, if the teachers can comment on each other's models, suggest improvements or come up with new ways of modelling the same features it could hopefully create an environment where the students are encouraged and more comfortable to do the same. And even if they do not engage in a discussion with the teachers it provides an opportunity for the students to *"think about the topic rather than memorize information"* [11].

It is easier for the teacher who is not actively involved in the discussion at a certain point in the lecture to interpret comments/questions from the students and to sense when the direction of the discussion changes. It therefore becomes easier to discuss alternatives when two teachers are present since we can help each other to interpret and integrate student ideas into the model being developed. At the same time, two teachers have a better chance of keeping the momentum of the lecture, ensuring that the objectives of the lecture are met.

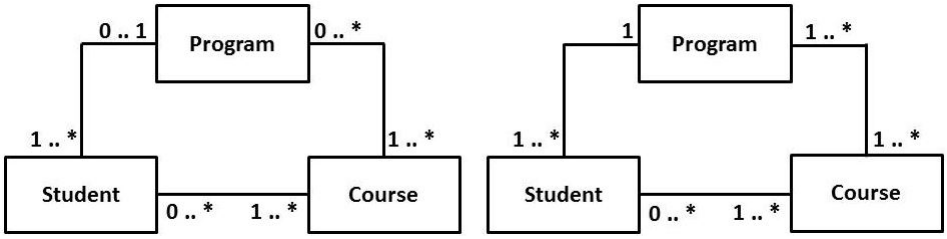

Figure 12: Two alternative ways of modelling the associations between three educational concepts.

In order to encourage students to actively participate in the lectures we used a real-life example of a software model where students had sufficient background knowledge: a system for registering students to courses. This example also serves as an illustration of what we mean by a *model* in the context of this paper. In Figure 12 there are two examples of a simplified software model, the domain model [14]. It depicts three central concepts within the educational domain (students, programs and courses) and how they relate to each other. In the left model a student can be registered to a programme at the most but it is not necessary (shown by the 0..1 notation). This is the situation for those registered as students at the University of Gothenburg. In the domain model to the right each student has to belong to a program (shown by a single 1). This reflects the situation at Chalmers University of Gothenburg. At Chalmers each course is also part of at least one program (1..*) while the University of Gothenburg allows courses that are not part of a program (0..*).

Neither solution is right nor wrong; the two models represent two different definitions of what roles programs really play and how that effects the associations to courses and students. These differences will percolate through the rest of the design and have consequences for how other concepts and associations can be defined, such as which courses that are available for a student or if we need to include universities as a concept or not. If one of the models is given as the (simplified) example of a domain model it is easy to take the concepts for granted. On the other hand, if the students have to come up with the concepts and define them during the lecture the different assumptions and alternatives are made explicit. And it becomes essential to motivate the different solutions.

In the above example, the students already have the perspective of domain experts and end-users while the aim of the lecture is to help them to take the perspective of developers and clients. With this kind of setup student participation is essential for driving the lecture forward.

To briefly summarize, in pair lecturing each lecture had well defined learning outcomes but followed a semi-structured script and contained no slides as the lecturing duo tried to model and verbalize the entire process of developing real-life software models. The models are developed and refined through the interaction between the students and the teachers in a fashion that resembles how software models are developed in an authentic setting.

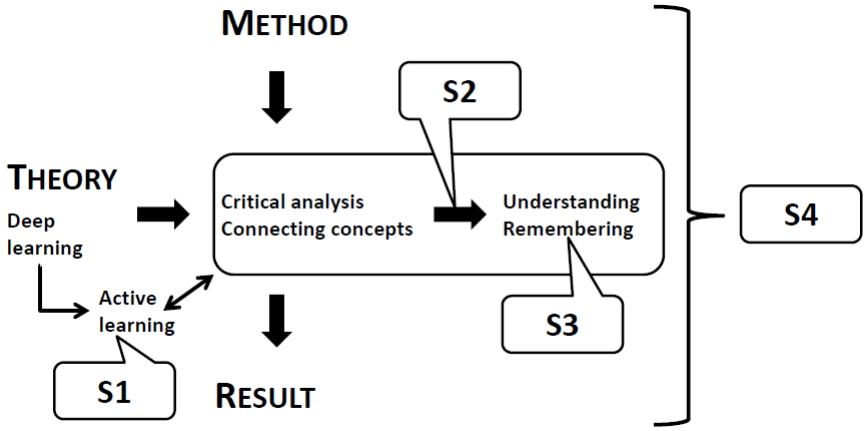

Figure 13: Aligning method, theory and results.

3 Results

The educational impact of pair lecturing was evaluated using a survey-based instrument. The survey contained a number of statements on a 5-point Likert scale: students circling 1 fully disagreed with a statement while those that circled 5 fully agreed with the statement. The survey also contained one open question: *What was the greatest value of pair lecturing?*

The statements and their relationship to method, theory and results are shown in Figure 13[8] where the four statements of the questionnaire were;

S1 *I was more active during the lectures than in traditional lectures.* The statement identifies to what extent the students felt more active, and therefore to what extent they took a deep approach to learning.

S2 *Pair lecturing lets the students influence the content of the course more than traditional lectures.* Here we query to what extent the students felt that they could influence the lecture content. The idea is that the students should be able to interact with the teachers to obtain the help they need in order to make the transition from critical analysis and connecting concepts to understanding and remembering.

S3 *I remember more after a lecture with pair lecturing than after a traditional lecture.* This statement probes to which extent pair lecturing leads to an increase in students' long-term remembering.

S4 *Pair lecturing should be used in more courses.* While S1 to S3 assured that the results were aligned to the method and theory S4 was introduced to assess if

[8]Compared to the original publication this section is extended with Figure 13 and belonging explanations.

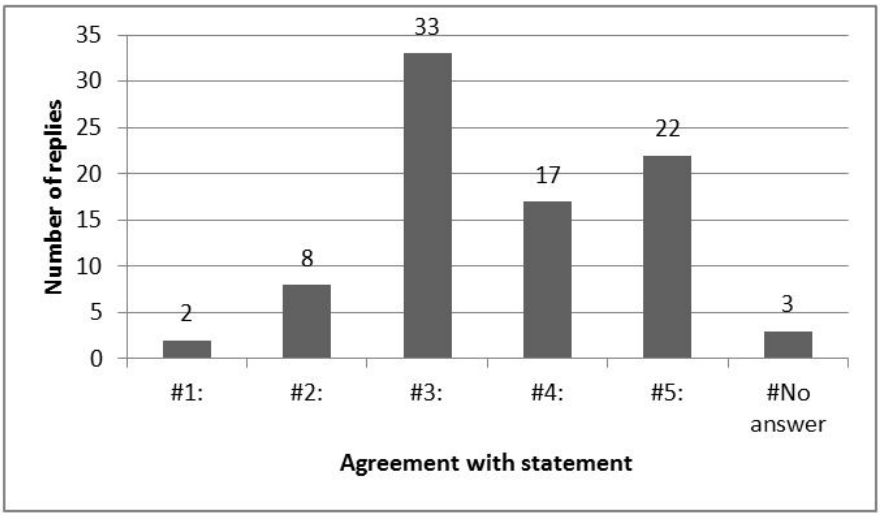

Table 1: S1: *I was more active during the lectures than in traditional lectures.*

cognitive apprenticeship through pair lecturing is in contradiction to what the students prefer as learning environment. This is important since a change in the educational setup should not harm the students' learning opportunities.

In regards to statement S1-S3 a traditional lecture was defined as a lecture where one teacher relies mainly on ready-made slides to deliver the lecture content.

The answers to the first statement, *I was more active during the lectures than in traditional lectures* are presented in Table 1. In this survey a traditional lecture was defined as one teacher relying mainly on slides to present the lecture content. Two students fully disagreed with the statement while 22 fully agreed. The mean score was 3.6. Concerning the value of pair lecturing one student wrote that *"it feels easier to ask questions"*. Other students claimed that *"the lectures come to life with the discussion, the arguing and the class interaction"* and that *"pair lecturing helps you stay focused on what the teachers are doing"*. All in all, 39 out of 85 students claimed that they were more active during pair lecturing than they usually are while 10 students did not agree.

The statement in Table 1 does not define *active*. To be more precise, we were interested in a more flexible lecture structure to let the students influence the lecture content. A question reflecting this issue is presented in Table 2. If we address what the students find interesting and challenging, this will hopefully help them to find their own way to accommodate the new knowledge. In addition, we wanted to create a situation where the students had to actively reflect upon the different ways to use the software models. Hopefully this implies that they will be more focused and remember more from the lectures. This aspect of active learning is addressed in Table 3.

In Table 2 the reply frequencies are shown for the statement *Pair lecturing lets the students influence the content of the course more than traditional lectures.* The mean

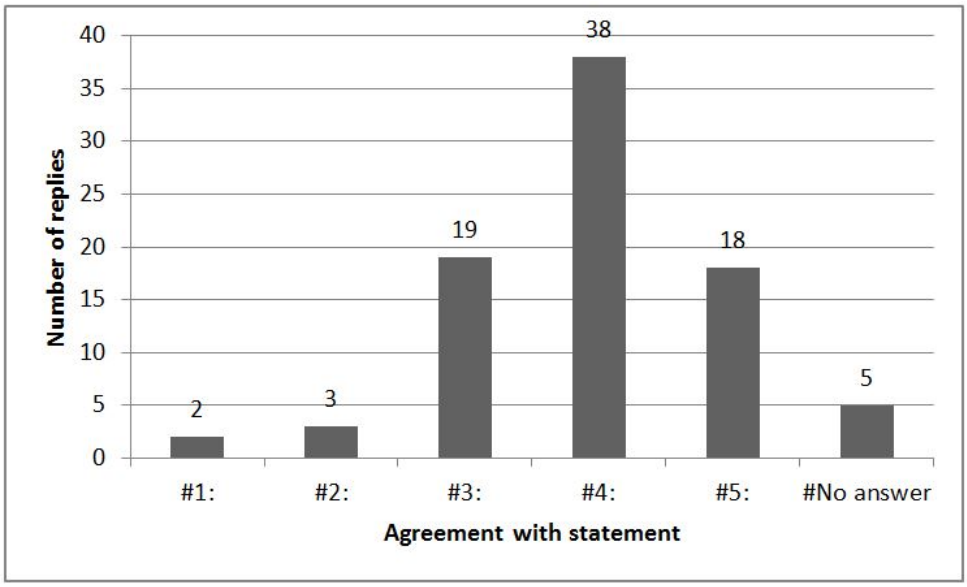

Table 2: S2: *Pair lecturing lets the students influence the content of the course more than traditional lectures.*

score was 3.8, with 18 students fully agreeing with the statement, and 56 out of 80 replies were on the positive side. There were no comments on this statement.

If there were no comments for the previous statement the situation was different for the statement *I remember more after a lecture with pair lecturing than after a traditional lecture*. One of the comments was that *"it is more interesting seeing two opinions, it gives a deeper (instead of broader) understanding"* while another student claimed that *"complicated things got explained twice"*. Many students appreciated that the teachers had different opinions about the models but a few stated that it became confusing and that it requires more reflection on their behalf when there are conflicting opinions about the models and how they should be used: *"while different views can be enlighting, it sometimes causes confusion"*. 55 students out of 82 claimed that they remembered more from the lectures than they usually do. The mean score for this statement was also 3.8.

To get an overall picture of what the students thought about pair lecturing we asked them to what extent they agreed with the following statement: *Pair lecturing should be used in more courses*. 33 students replied that they fully agreed. One of the comments was that *"I realized that the subject requires that you develop your own point of view"*. Another student had grasped the process from the lectures and wrote: *"I could use it to improve how I interact with my project team"*. Again the mean score was 3.8.

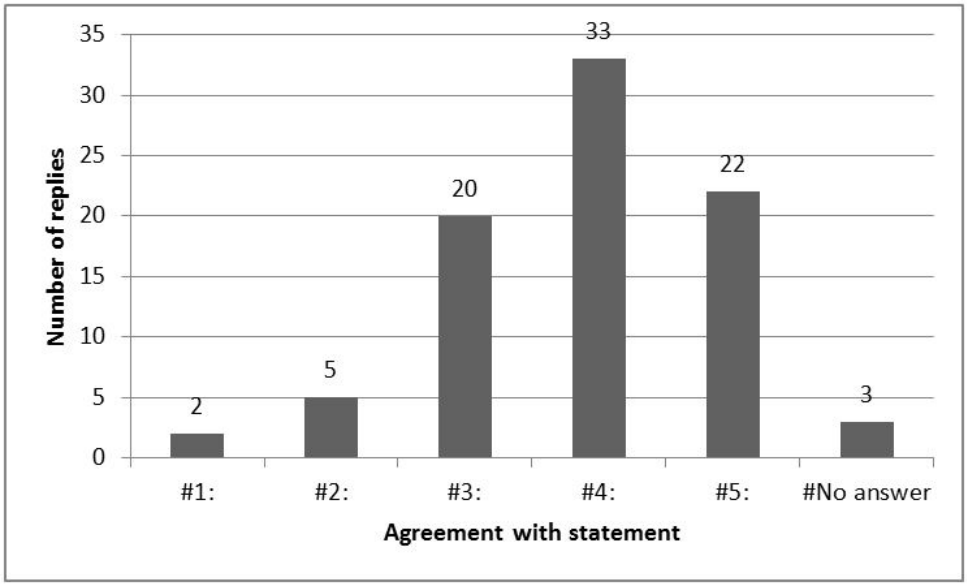

Table 3: S3: *I remember more after a lecture with pair lecturing than after a traditional lecture.*

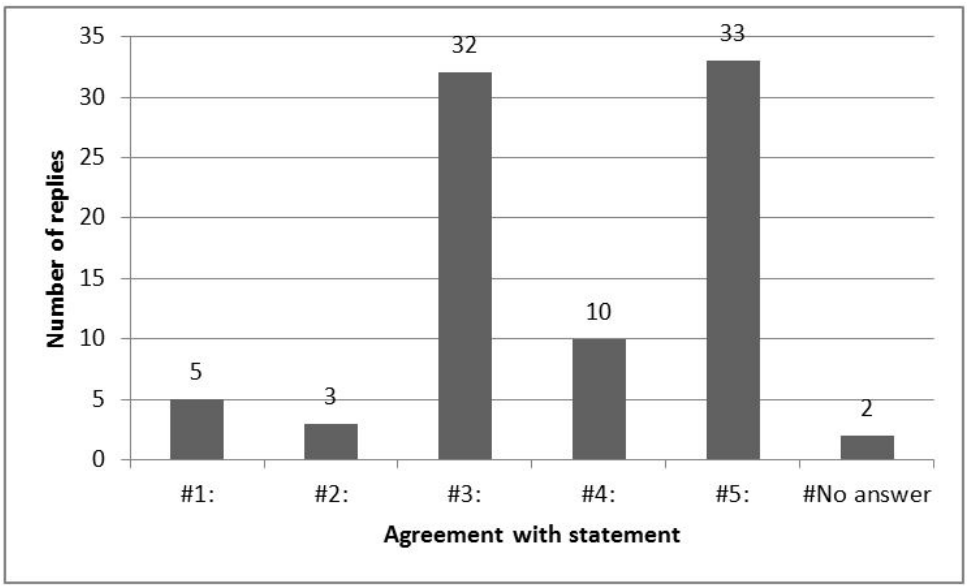

Table 4: S4: *Pair lecturing should be used in more courses.*

4 Discussion

We are encouraged by the fact that 55 out of 82 students said that they remember more after a lecture with pair lecturing than after a traditional lecture. Moreover, 40% of the students fully agreed that pair lecturing should be used in more courses. If we add those that agreed to some extent, over 50% agreed. On the other end of the scale we only have 10% who disagree. In fact, there are fewer students who disagree (8) than there were students failing the course (15).

Each lecture had well-defined learning objectives while still being flexible enough to adjust to the students' questions. However, some of the students felt that the lectures were unstructured; *"I was more active but lectures lacked structure"* and *"I think you should have a clear plan"*, were two comments. This might be a consequence of having to deal with several different viewpoints at the same time, as indicated in these comments: *"Try to agree more often"* and *"I found it difficult to understand important points when opinions differed too often"*. These comments are better understood in light of a scheme of cognitive development described by Perry [17]. According to Perry, college students move through a sequence of stages in which they hold different views on the nature and acquisition of knowledge. In his scheme, there is a progression from a simple dualistic view of knowledge, in which there is always one right answer, to a view that can embrace a multiplicity of viewpoints. *"Learning does not just affect what you now; it can transform how you understand the nature of knowing"* [1]. It is therefore important to expose students to different viewpoints (and uncertainty) to help them to realize that there can be more than one solution to a problem [16], and hence to facilitate cognitive development. But at the same time we do not want to make the lectures more complicated than necessary. Getting the balance right is not trivial and will take time.

Active learning can sometimes take students out of their comfort zone. While some students enjoy that teachers care enough to do something different, others get stressed when they are expected to learn a new subject in a new way. As Felder and Brent [8] point out, it is therefore essential to explain why you are using a new and non-traditional teaching method. Since we now have some experience of pair lecturing it will be easier to explain to the next group of students what we want to achieve and how it works.

While one teacher was busy drawing or explaining a model the other one had time for reflection: What are the practical implications for the students? What is missing in the drawing or explanation? How does our lecture fit in with industrial practice or the literature? This meant that we could give two perspectives or explanations for the same model or phenomena. For us this was an important gain since it is often easier for the teacher who has been listening to understand questions and comments from the students and then improve or elaborate on the answer given by the other teacher. When teaching alone it is very difficult to find the time to reflect - in real-time - on what you are saying and doing in the classroom. In his seminal book, *"The reflective practitioner"*, Schön [19] introduces the twin notions of reflection-in-action and reflection-on-action. Reflection-in-action can be described as *thinking on your feet* while reflection-on-action is done later, after the lecture. It is only after the lecture that you usually have the time to take a step back and reflect on what really happened

during the lecture; to evaluate the events and the decisions you made. And, in our experience, it is often only after the lecture that you fully understand the meaning of a student question or comment, and what a more appropriate answer would have been. But that teachable moment, that just-in-time opportunity to connect with the students is lost. One of the greatest benefits of pair lecturing is therefore the potential to enhance reflection-in-action. It also means that by being two teachers in the lecture room we do not only create more opportunities for teacher-student interaction, as expressed by Little and Hoel [15], we also make better use of the interaction.

The teacher that did not prepare the lecture could use the lecture itself as recapitulation of the models, reducing the time spent on preparing for supervision. This had the added benefit that each teacher knew what the other teacher had said during the lectures. It also increased the status of the teacher that earlier was mainly involved as a course assistant.

We think that it is important to ground pair lecturing in an existing collaboration. We used the same mutual reasoning and interaction with the students as we have in our own research to come up with suitable models for our running example systems. It also meant that we were comfortable in not having a fixed lecture plan since we knew from experience that we can handle these kinds of situations. One student commented in the survey that *"you complement each other"* which we believe is an important factor for pair lecturing to work.

We were initially concerned that we would have to lower the tempo in the course since drawing a model takes longer time than showing it on a slide. It turned out that we could keep the same tempo as previous years. It does take longer time to draw the model but this is compensated by the fact that we could introduce many of the details on-demand instead of showing a new slide with corresponding information. As a result the lectures were driven by the students' questions. At the same time we think that we managed to keep the lecture structured enough to cover what the students needed to know to get started with their own modelling.

There were two occasions when we opted to use slides. The first occasion was to show how the information encapsulated in several complicated models that were developed in previous lectures could be used when developing a new type of model that enhanced the information while still being consistent. This would have taken more time than the designated lecture time to draw by hand. The second slide show used the animation property to show how an instance of a model changes over time. We feared that the redrawing of a model would make the presentation messy and there is no easy way of rewinding the changes when recapitulating the process.

Many of our old examples relied on showing a new slide for each new aspect or detail that we wanted to discuss. When drawing the models this is not feasible so we had to come up with new examples that were reusable throughout a lecture. This meant an increase in preparation time this time around that we will not have next year, since the new examples will be reusable next time we give the course. So, changing lecturing style had the additional effect of enforcing a more stringent and efficient way of showing examples and presenting the course content.

5 Conclusion

The aim of this study was to answer two questions:

1. Can pair lecturing encourage students to take a deep approach to learning in lectures?

2. What are the pros and cons of pair lecturing for students and teachers?

By a deep approach to learning we follow the definition of Houghton [12] meaning that the students should be able to critically analyze new ideas and connect them to what they already know, which in turn leads to an increased understanding and long-term retention of the concepts.

To answer our first question the students were asked to comment on three separate statements, reported in Tables 1 to 3. From Table 1 we can conclude that pair teaching enabled the students to be more active during lectures, thus encouraging critical analysis and connecting new ideas to what they already have learnt. For the students to transform their new insights into a deeper understanding there has to be opportunities for them to reason about issues they find challenging during the lectures. The possibility for the students to influence the course content was reported on in Table 2, where the students state that there were more opportunities to influence content than usual. Finally, in Table 3 the students report that pair teaching helped them in remembering more of the lecture content than they usually do.

In answer to the second question we will first consider the student aspect. The negative comments from the students were about too many opinions confusing more than helping to understand the course subject. The positive comments mentioned complicated aspects being explained twice, better focus during lectures and deeper understanding. To determine if pair teaching is in conflict with what the students perceive as a constructive learning environment we asked them to comment if they recommend the use of pair teaching in more courses, Table 4. The response from the students regarding pair teaching was overall positive. As teachers we experienced a sensation of sticking out our necks a bit, but being two teachers in the lecture hall made us more comfortable and confident. Our way of doing pair teaching has benefitted from the fact that we have an established research collaboration, we know each other's strengths and weaknesses as well as how to cooperate and tackle challenges together. The increase in teaching hours was compensated by an equivalent cut in preparing for supervision, though there was an increase in time for preparing the lectures since our old examples had to be adjusted to fit with our new lecture plans. We found that pair teaching gave us a better opportunity to reflect *in-action*, which enabled us to handle student initiatives in a constructive way to drive the lectures forward. All in all, as educators we found pair teaching to be a positive experience. The simple fact is that lecturing is much more fun in pairs.

Bibliography

[1] Ken Bain. *What the Best College Teachers Do*. Harvard University Press, Cambridge, MA, 2004.

[2] John Burville Biggs. *Teaching for Quality Learning at University : What the Student Does*. Society for Research into Higher Education : Open University Press, 2nd edition, 2003.

[3] C.C. Bonwell and J.A. Eison. *Active learning: creating excitement in the classroom*. ASHE-ERIC higher education report. School of Education and Human Development, George Washington University, 1991.

[4] John Seely Brown, Allan Collins, and Paul Duguid. Situated cognition and the culture of learning. *Educational Researcher*, 18(1):32–42, Jan-Feb 1989.

[5] F.J. Buckley. *Team Teaching: What, Why, and How?* Dissertation and Proposal Writing Series. SAGE Publications, 2000.

[6] Håkan Burden, Rogardt Heldal, and Tom Adawi. Assessing individuals in team projects: A case study from computer science. In *Conference on Teaching and Learning - KUL*, Gothenburg, Sweden, January 2011.

[7] Håkan Burden, Rogardt Heldal, and Toni Siljamäki. Executable and Translatable UML – How Difficult Can it Be? In *APSEC 2011: 18th Asia-Pacific Software Engineering Conference*, Ho Chi Minh City, Vietnam, December 2011.

[8] R. M. Felder and Brent R. Learning by Doing. *Chemical Engineering Education*, 37(4):282–283, 2003.

[9] R. M. Felder and Brent R. Active Learning: An Introduction. *ASQ Higher Education Brief*, 2(4), August 2209.

[10] Richard R. Hake. Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *American Journal of Physics*, 66(1):64–74, 1998.

[11] F Hanusch, L Obijiofor, and Z Volcic. Theoretical and Practical Issues in Team Teaching a Large Undergraduate Class. *International Journal of Teaching and Learning in Higher Education / Vol. 21, No. 1, pp.66-74*, 21(1):66–74, 2009.

[12] Warren Houghton. *Engineering Subject Centre Guide : learning and teaching theory for engineering academics*. Teaching Guides (HEA Engineering Subject Centre). Higher Education Academy Engineering Subject Centre, Loughborough University, 2004.

[13] David Kember and Lyn Gow. Action research as a form of staff development in higher education. *Higher Education*, 23(3):297–310, 1992.

[14] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[15] A. Little and A Hoel. Interdisciplinary Team Teaching: An Effective Method to Transform Student Attitudes. *The Journal of Effective Teaching*, 11(1):36–44, 2011.

[16] L.B. Nilson. *Teaching at its Best*. Jossey-Bass, San Francisco, CA, 3rd edition, 2010.

[17] W.G. Perry. *Forms of Intellectual and Ethical Development in the College Years: A Scheme*. Bureau of Study Counsel, Harvard University, MA, 1968.

[18] Michael Prince. Does Active Learning Work? A Review of the Research. *Journal of Engineering Education*, 93(3):223–231, 2004.

[19] D.A. Schön. *The Reflective Practitioner: How Professionals Think in Action*. Number TB5126 in Harper Torchbooks. Basic Books Publ., 1983.

[20] J. Stice. *Teaching problem solving*. TA handbook. Texas University, TX, 1996.

Appendix B:
Scholarship of Discovery

Paper 3:
Language Generation from Class Diagrams

Published as:
Natural Language Generation from Class Diagrams
Håkan Burden[1] and Rogardt Heldal[1]
[1] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
MoDeVVa 2011
Model-Driven Engineering, Verification and Validation
Wellington, New Zealand
17 October 2011

1 Introduction

In Model-Driven Architecture (MDA; [14]) software models are transformed into code in a series of transformations. The models have different purposes and level of abstraction towards the resulting implementation.

A Computational Independent Model (CIM) shows the environment of the software and its requirements in a way that can be understood by domain experts. The CIM is often referred to as the domain model and is specified using the vocabulary of the domain's practitioners and the stakeholders [16].

In the transformation from a CIM to a Platform Independent Model (PIM) the purpose of the models change and the focus is on the computational complexity that is needed to describe the behaviour and structure of the software.

The PIM is then transformed into a Platform Specific Model (PSM) which is a concrete solution to the problem as specified by the CIM. The PSM will include information about which programming language(s) to use and what hardware to deploy the executable code on.

One way of realising the model transformations in the MDA process is shown in Figure 14 which is adopted from [16]. In this process the transformation from CIM to PIM is done manually while the transformation from PIM to PSM is formalised by using marks and mappings. The marks reflect both unique properties of a certain PSM as well as domain-specific properties of the PIM, while the mappings describe a model to model transformation [14].

In MDA the PIM should be a bridge between the CIM and the PSM. Thus it is important that the PIM is clear and articulate [10, 27] to convey the intentions and motivations in the CIM as well as correctly describe the PSM [20].

1.1 Motivation

The developers of the PIM have to interpret the CIM to make their design decisions. Thus there are many ways for the PIM to represent a different solution to the problem compared to the solution given by the CIM: The CIM might be ambiguous or use vaguely defined concepts with the risk that it is misinterpreted; the CIM might be incomplete in the view of the developers of the PIM so they make additions to the PIM and finally, the CIM might be assessed as incorrect but the correction is made in the PIM and not in the CIM. Over time the CIM and the PIM diverge due to the interaction of these inconsistencies.

The problem is not limited to the development phase. In order to adopt the CIM and the PIM to changing requirements, new developers have to be able to understand why the models are designed in the way they are and how they can be changed according to their underlying theory [17].

An example of the threat of failing to understand the underlying theory is given in [2]. From their experiences at British Airways they report on how important business rules are trivialised in the PIM as it is incapable of showing which business requirements are most important when all elements look the same in a class diagram. To demonstrate their point they use the notion of codesharing. Codesharing is when airlines in an alliance can sell seats on each others flights. For this to be possible

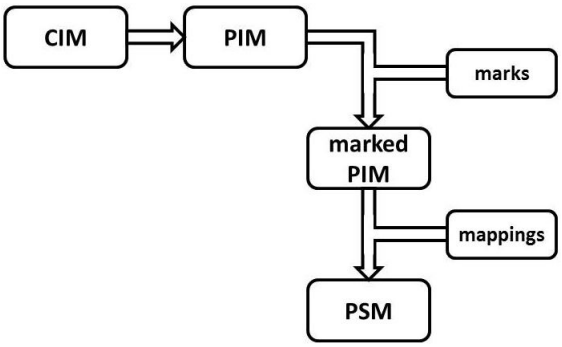

Figure 14: One realisation of the MDA process

a flight has to be able to have more than one flight code. In a class diagram this business requirement worth millions of pounds is obscured as a simple multiplicity on an association between two classes, see Figure 15.

So the transformation from CIM to PIM poses two questions: How do we know that the PIM captures the requirements of the CIM, and nothing else? And how can we make sure that future developers of the PIM understand the intentions and motivations behind the design decisions [17]? The evaluation of the correctness of the PIM's behaviour and structure can be done by testing and model reviewing.

Both testing and accessing the information of the PIM requires an understanding of object-oriented design, knowledge of the used models and experience of using tools for software modelling [2]. Textual descriptions, on the other hand, are suitable for stakeholders without the necessary expertise in software models [8]; natural language can be understood by anyone, allowing all stakeholders to contribute to the validation of the PIM.

1.2 Aim

Our long-term aim is to reverse engineer the marked PIM into a CIM, investigating how much of the original CIM that can be generated from the marked PIM. As our first step towards a complete system we have chosen the structure of the class diagrams. The aim of the generated text is not only to paraphrase the class diagram but also to include the underlying motivations and design decisions that form the theory behind the model.

By using an MDA approach for generating natural language text we enable the textual description of the PIM, the PIM itself and deployed PSMs to be synchronised with each other. The texts can be used by stakeholders that are unfamiliar with software models to validate the structure and behaviour of the models, enabling a process that leads to software meeting the requirements and expectations of all stakeholders.

1.3 Contribution

We have generated textual descriptions of the structure of the class diagram that not only paraphrase the diagrams but also include the underlying motivations and design decisions. The mappings from marked PIM to natural language PSMs are generic and can be applied to any marked PIM. Indeed, since the marks are used to enhance the performance of the mappings the transforming an unmarked PIM will still generate a linguistic model. Though the text generated from such a linguistic model might have minor grammatical errors.

The vocabulary of the PIM is reused as lexicon for the generated linguistic model so that we can generate text for any domain independent of how technical or unpredictable the vocabulary may be.

In MDA terms the generation of natural language was solved by first transforming the xtUML models into an intermediate linguistic model, a grammar. In a second transformation the grammar was used to generate the desired view of the class diagrams as natural language text.

1.4 Overview

In the next section we present the background knowledge for our case study in terms of natural language generation, the Grammatical Framework and Executable and Translatable UML. In section 3 we describe our case study of transforming the PIM into a CIM. The results are given in section 4, followed by a discussion in section 5. Our case study is related to previous work in section 6 and a summary with drafts for future work concludes our contribution.

2 Background

In our case study we have used the MDA perspective on models for Natural Language Generation [23]. This was achieved by first transforming the marked PIM into a linguistic model defined by the Grammatical Framework [22]. The linguistic model was then used to generate the final textual description of the PIM. We used Executable and Translatable UML to model the class diagram and the model to model transformation.

2.1 Executable and Translatable UML

The Executable and Translatable Unified Modeling Language (xtUML; [13, 21, 28]) evolved from merging the Shlaer-Mellor method [24] with the Unified Modeling Language, UML[9].

There are three kinds of diagrams used in xtUML (component diagrams, class diagrams and statemachines) as well as a textual Action language. The Action language is used to define the semantics of the graphical diagrams. This study only concerns the class diagrams.

[9] `http://www.omg.org/spec/UML/2.3/`

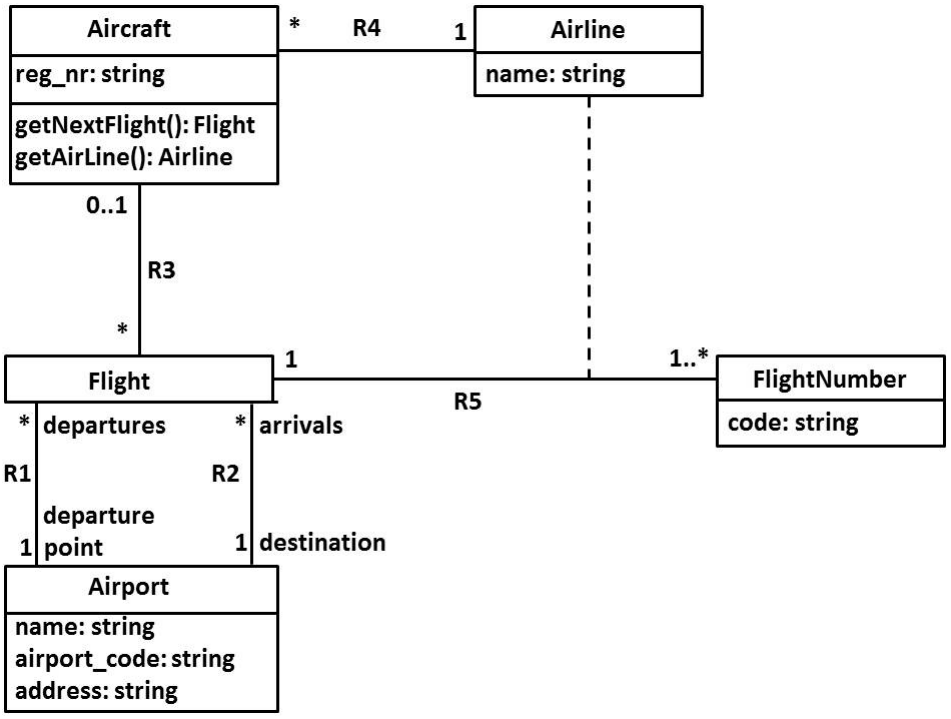

Figure 15: Our example class diagram

2.1.1 xtUML Class Diagrams

In Figure 15 we have an example of an xtUML class diagram. The xtUML classes and associations are more restricted than in UML. We will only mention those differences that are interesting for our case study.

In UML the associations between classes can be given a descriptive association name while in xtUML the association names are automatically given names on the form `RN` where `N` is a unique natural number. I.e. `Flight` is associated to `FlightNumber` over the association `R5`.

In xtUML there are no special associations for the UML aggregate and composition associations. Both aggregation and composition express a parts-of relation with the difference that in aggregation, the parts can exist without a 'whole' while in composition the parts cannot exist without the 'whole'. Following the definition given by the OMG[10] aggregation is modelled by using the multiplicity 0..1 and composition by using the multiplicity 1.

Speaking of multiplicities, in xtUML there are only four possible combinations of multiplicities; 0..1, 1, * and 1..*.

[10] `http://www.omg.org/spec/UML/2.3/`

2.1.2 Model Transformation

The PIM to PSM transformation is handled by model compilers. A model compiler takes a marked PIM and a set of mappings that specify how the different elements of the marked PIM are to be translated into the PSM [14, 16]. Since the PSM is generated from the marked PIM, it is possible for the running code and the software models to always be in synchronization with each other since all updates and changes to the system are done at the PIM-level, never by touching the PSM. The model compiler allows the same PIM to be transformed into different PSMs [1] without a loss in efficiency compared to handwritten code [25].

2.2 Natural Language Generation

When compiling a marked PIM into a PSM it is important to include all the information of the marked PIM into the transformation. For Natural Language Generation (NLG) this is not the case [23]. The content, its layout and the internal order of the generated text is dependent on who the reader is, the purpose of the text and by which means it is displayed. In this sense the texts can be seen as platform-specific.

Traditionally NLG is broken down into a three-stage pipeline; text planning, sentence planning and linguistic realisation [23]. From an MDA perspective NLG can be viewed as two transformations. The first transformation takes the software model and reshapes it to an intermediate linguistic model by performing text and sentence planning. The second transformation is equivalent to the linguistic realisation as the linguistic model is transformed into natural language text. We will use our class diagram in Figure 15 to exemplify the purpose of the three stages.

2.2.1 Text Planning

Text planning is to decide on what information in the original model to communicate to the readers. When the selection has been done the underlying structure of the content is determined. In our case we first describe the classes with attributes and operations, then the associations between the classes with multiplicities.

2.2.2 Sentence Planning

When the overall structure of the text is determined the attention is turned towards the individual sentences. This is also the time for choosing the words that are going to be used for the different concepts, e.g. an aircraft can both *depart* or *leave* an airport. The original software model has now been transformed into a linguistic model.

2.2.3 Linguistic Realisation

In the last stage the linguistic model is used to generate text with the right syntax and word forms. The linguistic model should ensure that the nouns get the right plural forms and that we get ***a*** *flight* but ***an*** *aircraft*. Through the linguistic realisation the intermediate model has been transformed into a natural language text.

2.3 Grammatical Framework

For defining the linguistic model we use Grammatical Framework (GF, [22]). In GF the grammars are separated into an abstract and a concrete syntax. To understand how we have used GF and the resource grammars we give an example that generates the sentence *An Aircraft has many Flights*. The grammar is found in Figure 16. It is not necessary to understand the details of the grammar, it is included as a small example of the kind of output that is generated from our model to model transformation.

2.3.1 Abstract Syntax

The abstract syntax is defined by two finite sets, categories (`cat`) and functions (`fun`). The categories are used as building blocks and define the arguments and return values of the functions

From the class diagram in Figure 15 we have that both `Aircraft` and `Flight` are class names. We want to use this information in our grammar, defining a function for both `Aircraft` and `Flight`, see Figure 16. From a linguistic point of view they define the lexical items that make up our lexicon. Lexical items can be used to define more complex functions, like `OneToMany` that returns a `Text` describing the association between two `ClassName`s. By defining our categories (the content of the text) and the functions (the ordering of the content) we have completed the text planning stage of the NLG process.

2.3.2 Abstract Trees

Abstract syntax trees are formed by using the functions as syntactic constructors according to their arguments. While the abstract syntax shows the text planning for a possibly infinite set of texts the abstract tree represents the structure of exactly one text. According to our example grammar the sentence *An Aircraft has many Flights* will have `OneToMany(Aircraft, Flight)` as its abstract tree.

2.3.3 Concrete Syntax

A concrete syntax assigns a linearisation category (`lincat`) to every abstract category and a linearisation rule (`lin`) to every abstract function. The linearisation categories define how the concepts of the PIM are mapped to the pre-defined categories of GF. From an NLG perspective the linearisation rules supply the sentence planning. The concrete syntax is implemented by using the GF Resource Grammar Library.

2.3.4 Resource Grammar Library

In the Resource Grammar Library (RGL) a common abstract syntax has sixteen different implementations in form of concrete syntaxes. Among the covered languages are English, Finnish, Russian and Urdu. The resource grammars come with an interface which hides the complexity of each concrete language behind a common abstract interface.

Abstract syntax:

```
cat Text, ClassName ;

fun Aircraft : ClassName ;
    Flight : ClassName ;
    OneToMany : ClassName × ClassName → Text ;
```

Concrete syntax:

```
lincat Text = RGL.Text ;
       ClassName = CN ;

lin Aircraft =  mkCN (mkN "Aircraft" "Aircraft") ;
    Flight = mkCN (mkN "Flight") ;
    OneToMany aircraft flight =
      mkText (mkCl (mkNP (mkDet a_Quant) aircraft)
             (mkV2 have_V)
             (mkNP (mkDet many_Quant) flight)) ;
```

Figure 16: An example of an automatically generated GF grammar

The RGL interface supplies a grammar writer with a number of functions for defining a concrete syntax. In Figure 16 `mkText`, `mkCl` and `a_Quant` are examples of such functions. Exactly how these functions are implemented is defined by the concrete resource grammar for each language. Just as for a programming language we only need to understand the interface of the library to get the desired results, we do not need to understand the inner workings of the library itself.

2.3.5 Linearisation

In GF the linearisation of an abstract tree, $\mathtt{t}$, by a concrete syntax, $\mathtt{C}$, can be written as $\mathtt{t}^{\mathtt{C}}$ and formulated as follows

$$(\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n))^{\mathtt{C}} = \mathtt{f}^{\mathtt{C}}(\mathtt{t}_1^{\mathtt{C}}, \ldots, \mathtt{t}_n^{\mathtt{C}})$$

where $\mathtt{f}^{\mathtt{C}}$ is a concrete linearisation of a function $\mathtt{f}$ [12].

The linearisation of `OneToMany(Aircraft, Flight)` using the concrete English grammar `ENG` described in Figure 16 is then unwrapped as follows

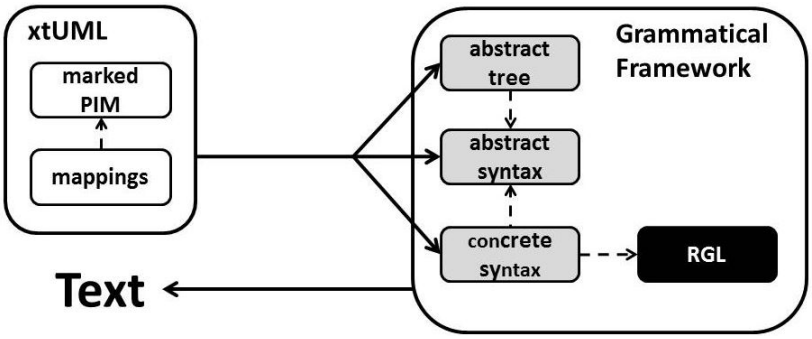

Figure 17: From marked PIM to text

$$
\begin{aligned}
&(\texttt{OneToMany}(\texttt{Aircraft}, \texttt{Flight}))^{\texttt{ENG}} \\
&\quad = \texttt{OneToMany}^{\texttt{ENG}}(\texttt{Aircraft}^{\texttt{ENG}}, \texttt{Flight}^{\texttt{ENG}}) \\
&\quad = \texttt{mkText}(\texttt{mkCl}(\texttt{mkNP}(\texttt{mkDet a\_Quant})\ \texttt{Aircraft}^{\texttt{ENG}}) \\
&\quad\ (\texttt{mkV2 have\_V}) \\
&\quad\ (\texttt{mkNP}(\texttt{mkDet many\_Quant})\ \texttt{Flight}^{\texttt{ENG}})) \\
&\quad = \texttt{mkText}(\texttt{mkCl}(\texttt{mkNP}(\texttt{mkDet a\_Quant}) \\
&\quad\ (\texttt{mkCN}(\texttt{mkN "Aircraft" "Aircraft"})))) \\
&\quad\ (\texttt{mkV2 have\_V}) \\
&\quad\ (\texttt{mkNP}(\texttt{mkDet many\_Quant})(\texttt{mkCN}(\texttt{mkN "Flight"})))) \\
&\quad = \textit{An Aircraft has many Flights}
\end{aligned}
$$

Linearisation is an built-in functionality of GF and equivalent to the linguistic realisation of NLG.

3 Natural Language Generation from Class Diagrams

To investigate the possibilities for natural language generation from software models we have conducted a case study using xtUML to model the PIM and perform the model-to-model transformations. The reason for choosing xtUML is that the model compiler enables a convenient way of transforming the PIM to different PSMs. We used BridgePoint[11] [13] as our xtUML tool.

3.1 Case Description

The original case was a hotel reservation system. To avoid getting into domain details and explaining the different components and subsystems we reuse the example given in

[11] http://www.mentor.com/products/

[2] with a small extension; we have added classes for the concepts `Aircraft`, `Airport` and `Airline`. The result is a class diagram that highlights the problems we want to solve and what we can achieve in forms of NLG. The class diagram can be found in Figure 15. The intention of the diagram is not a complete description of the problem domain.

Our PIM includes a note for the association `R5`, *A Flight can have more than one Flight number since code sharing is a multimillion-pound business, affecting an alliance of airlines.* There are also notes on the associations so that they carry meaningful association names instead of xtUML's generic ones. `R1` and `R2` are annotated with *has*, `R3` is annotated with *is booked for*, `R4` is annotated with *belongs to* which is to be read from left-to-right only and `R5` has the note *is identified by* which also is to be read from left-to-right.

An overview of our system is found in Figure 17. The shaded modules are generated in the model-to-model transformation. The Resource Grammar Library (RGL) supplies the necessary details to realise the concrete syntax. The dotted lines within the systems give the dependencies between the modules while the solid lines show the transformations between the systems. The transformation between xtUML and Grammatical Framework is defined as mappings in BridgePoint while the transformation from Grammatical Framework to text is automatically handled by GF through linearisation.

The input to the first transformation in Figure 17 is a marked PIM and a set of mappings. The marks are described next and then the mappings.

3.1.1 Marking the PIM

Since we are aiming for a linguistic model and not source code we use marks for irregular word forms, where the marks play a similiar role as stereotypes in UML. In our example we use a mark on the class `Aircraft` so that the noun *Aircraft* has the same form in both singular and plural. Just as for UML the xtUML metamodel can be extended for different profiles. Our extension results in a natural language profile for xtUML. The general mapping is otherwise to use the regular form for English nouns, i.e. a plural s. The mappings are generic and can be used for any marked PIM.

3.1.2 Mappings

We use the pseudo-algorithm in Figure 18 to decide what the linguistic model should contain and in what order. These mappings are generic and can be used for any marked PIM. The mappings only consider certain aspects of the class diagrams of the PIM and if it contains other diagrams or action language this information is just omitted. The algorithm is implemented by using the xtUML model compiler.

3.2 xtUML to GF

3.2.1 Lexicon generation

Before we generate the different sentences of our text we need a vocabulary. The content of the vocabulary, or lexicon in linguistic terms, is taken from the names of

```
generate lexicon for class diagram;

for each class in class diagram
   if class has attributes
      generate sentence for class attributes ;
   if class has operations
      generate sentence for class operations ;

for each association in class diagram
   if association has association name
      generate sentences for association ;
   if association has association class
      generate sentence for association class ;
   if association has motivation
      generate sentence for motivation ;
```

Figure 18: The algorithm for generating the lexicon and grammar.

the elements of the class diagram and the marking model. The lexicon therefore defines which concepts that will be included in the final text (flights, names, codes etc.) and for which reason (as class names, attributes and so on). Here is the automatically generated abstract syntax of the lexicon, in a dense representation to save space.

```
cat ClassName, Association, Attribute,
    Multiplicity, Operation, Motivation ;

fun Flight, FlightNumber, Aircraft, Airline,
        Airport : ClassName ;
    R1, R2, R3, R4, R5 : Association ;
    Name, Code, RegNr, Address,
        AirportCode : Attribute ;
    One, ZeroOne, ZeroMore,
        OneMore : Multiplicity ;
    GetNextFlight, GetAirline : Operation ;
    R5Motivation : Motivation ;
```

3.2.2 Classes

To list the attributes of a class we generate a unique abstract function for each class with one `Attribute` argument for each class's attribute in the PIM. The function corresponding to the class `Airport` has the following abstract syntax

```
AirportAttributes : ClassName × Attribute ×
     Attribute × Attribute → Text ;
```

At the same time we generate an abstract syntax tree for the function given the class it paraphrases

```
AirportAttributes(Airport, Name,
                  AirportCode, Address)
```

The same procedure as for attributes is repeated for listing the operations of the classes.

3.2.3 Associations

We generate one function for all associations

```
Association : Association × Multiplicity ×
    ClassName × Multiplicity × ClassName →
    Text ;
```

This function is a generalisation of the `OneToMany` found in Figure 16. For the association between `Flight` and `Flight Number` we get the following tree

```
Association(R5, One, Flight, OneMore, FlightName)
```

To generate a text for an association class we use one function that takes three class names as arguments

```
AssociationClass : ClassName × ClassName ×
    ClassName → Text ;
```

For each association with an association class we then generate an abstract syntax tree. For association R5 in our example diagram we get the following tree

```
AssociationClass(Flight, FlightName, Airline)
```

Each motivation is introduced into the grammars by a unique function and abstract tree

```
R5Text : Motivation → Text ;
R5Text(R5Motivation)
```

3.2.4 Combining texts

We now have a set of unconnected abstract trees. To combine the trees into one text we introduce the function

```
Combine : Text × Text → Text ;
```

If we append the generated abstract trees above, we get the following abstract tree

```
Combine(
   AirportAttributes(Airport, Name,
                     AirportCode, Address),
   Combine(
      Association(R5, One, Flight,
                  OneMore, FlightNumber),
```

```
Combine(
   AssociationClass(Flight, FlightName,
                    Airline)
   R5Text(R5Motivation))))
```

We have now automatically transformed the class diagram into an abstract and a concrete syntax as well as an abstract syntax tree. Together these three represent a linguistic model of the text that we want to generate.

3.3 GF to Text

The generated abstract syntax tree for the document is linearised by the GF lineariser. The linearisation of the tree completes the transformation of our xtUML class diagram into natural language text.

4 Results

To show the results from our NLG process we give a small text that is generated from the examples used in the previous section.

An Airport has a name, an airport code and an address. An Aircraft can get next Flight and get Airline. A Flight is identified by one or more Flight Numbers. The relationship between a Flight and a Flight Number is specified by an Airline. A Flight can have more than one Flight number since code sharing is a multimillion-pound business, affecting an alliance of airlines.

The generated text can now be used by the stakeholders to validate that the class diagram has the right structure and that the underlying theory is represented. The generation of textual descriptions from the class diagram enables close communication with stakeholders, giving them constant feedback which is a crucial point according to [8].

The grammars were automatically transformed from the class diagram, all we needed to do was to mark the PIM and give the mappings between the marked PIM and the grammar. To generate text from another class diagram we need new marks for the irregular nouns. We can then reuse the mappings defined in our example to generate natural language text from any marked PIM.

Since the role of the marks is to enhance the quality of the transformation defined by the mappings it is not necessary to start with a marked PIM. The results of applying the mappings to an unmarked PIM is that we get a grammar treating all class names as regular nouns. This might lead to some odd phrasings, such as *many Aircrafts*. The division of labour between marks and mappings means that a developer with a reasonable knowledge of English can mark the PIM with the necessary irregularities while an expert on the target langauge and the used grammar formalism can define the mappings once and for all.

A further result is that we managed to combine two different systems that are successful within their respective domains. Executable and Translatable UML (xtUML) has previously been proven to allow the PIM and the PSM to be consistent with each

other as well as enabling reuse [1, 25]. GF is currently used in collaboration with industry for multilingual translation in the MOLTO-project[12] and has previously been used for multi-modal dialogue systems [3] and in collaboration with the car industry [11].

5 Discussion

In our Motivation we stressed that even a well-formed model is difficult to understand, thus the need for textual paraphrasing of its content and motivations. On the other hand, paraphrasing the model will not make up for a lack of detail in the model, those details are needed to make the text informative. It is therefore important that the models use meaningful names for classes, attributes and associations etc. so that it is possible to generate a precise vocabulary and meaningful descriptions of why classes are associated with each other.

In UML we can use verbs or verb phrases for the association names and nouns for the role names [15]. The role names can thus be seen as outsourced attributes. The problem is how to incorporate the information given by the class name and the role name together with the association name. For the class diagram in Figure 15, we state that *An Airport has one or more arrivals*. But what is an arrival? A clarification can be done in many ways, one is by adding subordinate clauses that define an arrival, *where an arrival is a Flight*. In xtUML the issue is solved differently.

Associations are given default names in xtUML, names that have no semantic meaning to a human reader. To understand what the association represents one has to understand the Action language that defines the association. The lack of a verb phrase for the association opens up a new way of looking at role names; [27] advocate that the role names should be used as underspecified verb phrases that are missing their complement. By using this definition of role names on our class diagram we get a new diagram adopted to xtUML, see Figure 19. The benefit is that we do not need to mark the associations to give them meaningful names and we can use the roles of the classes at the same time. From this diagram we could generate the sentence *An Airport has one or more arriving Flights*.

6 Related Work

In a systematic literature review from 2009 there is only one work that reports on generating natural language text from class diagrams. From our own searches we have not found any MDA approach that cites the review. However there are other contributions that have used the same techniques as we have, but in other settings.

A systematic literature review on text generation from software engineering models is reported in [18]. Of the 24 contributions only one concerned the generation of natural language text from UML diagrams, [15]. The motivation for conducting the literature review was that even if models are precise, expressive and widely understood by the development team natural language has its benefits. Natural language enables the

[12] `http://www.molto-project.eu/`

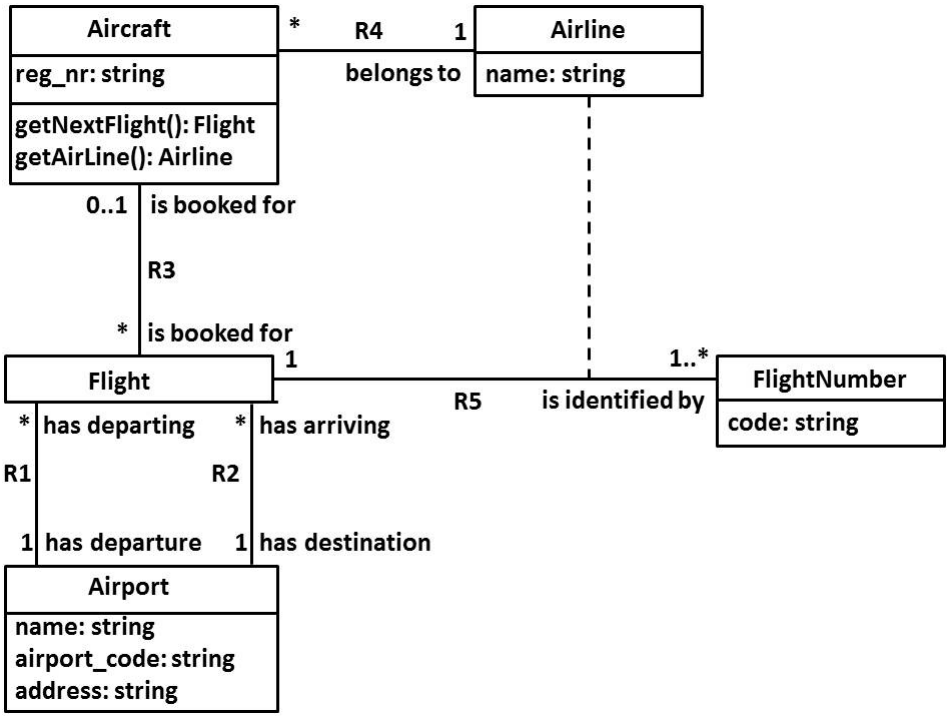

Figure 19: Our class diagram revisited

participation of all stakeholders in the validation of the requirements and makes it clear how far the implementation of the requirements have come. [18] state that none of the contributions address the issue of keeping the generated documents synchronized with the PIM.

Our examples of generated text are inspired by the work done by [15]. They generate natural language descriptions of UML class diagrams using WordNet [7] for obtaining the necessary linguistic knowledge. WordNet is a wide-coverage resource which makes it useful for general applications but can limit the use for domain-specific tasks. We use a domain-specific grammar that is tailored for just our needs. Whatever the domain our approach has lexical coverage while WordNet will lack lexcial knowledge about more technical areas. When it comes to results there texts are descriptions of the class diagram while ours also include the underlying motivations for the structure.

In [6] the Semantics of Business Vocabulary and Business Rules (SBVR, [19]) is used as an intermediate representation for transforming UML and OCL into constrained natural language. This means that SBVR maps to a limited set of possible sentence structures while GF allows a free sentence planning.

[4] have developed a system that transforms class diagrams into natural language texts. Their system differs from ours in that it marks all model elements with the corresponding linguistic realisation. While our system relies on the linguistic model

to perform the linguistic realisation, their system maps the marks straight into predefined sentences with slots for the linguistic realisation of the model elements.

Grammatical Framework has been used before to generate requirements specifications [5, 9] in the Object Constraint Language (OCL; [29]). GF is used to translate expressions in OCL to English text with LaTeX-formatting. The translation is done by implementing an abstract grammar for the UML model of OCL, a concrete grammar for OCL expressions and a concrete grammar for English. The text to text translation is then done by obtaining an abstract tree through parsing the OCL-expression, then linearizing the tree in English. Since we do not have a grammar for our graphical models we instead use the metamodel of xtUML to generate the necessary linearisation grammars.

7 Conclusions and Future Work

7.1 Conclusion

From our generated text it is possible to see if the motivations and intentions of the CIM are captured by the PIM. The texts also paraphrases the structure of the class diagram, enabling stakeholders with various backgrounds to participate in the validation of the PIM. In the process we have transformed the class diagram into an intermediate linguistic model which ensures that the generated texts are grammatically correct.

7.2 Future Work

From our case study we have identified two lines of future work that we find interesting. The first line is to generate other views of the PIM, the second line is to make more use of the Grammatical Framework.

So far we have looked at the static structure of the class diagram. Another aspect worth looking in to is the dynamic behaviour of the software. This can be done by transforming the Action language code into textual comments, adopting the results from [26] to xtUML and MDA. This will then be combined with natural language descriptions of the statemachines since they play a key role in the behaviour of objects.

There are several ways to make more use of GF. [5] enrich their generated texts with LaTeX, something that could be used to highlight the motivations or for supplying tags for colour and fonts to the texts. We also want to make more use of GF's capacity for several concrete languages to share the same abstract syntax. Being able to generate a variety of languages from internal system specifications would mean that the models can be accessed and evaluated by those stakeholders that are not confident in using English. One of the new languages could be a formal language for writing requirements and then GF could be used to both generate natural language descriptions, formal requirements and translate between the two.

Both lines of work will in the end require a more rigorous evaluation, both to obtain the desired format and content of the texts but also to see in which extent they can replace the original CIM.

Bibliography

[1] Staffan Andersson and Toni Siljamäki. Proof of Concept - Reuse of PIM, Experience Report. In *SPLST'09 & NW-MODE'09: Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, Tampere, Finland, August 2009.

[2] Jim Arlow, Wolfgang Emmerich, and John Quinn. Literate Modelling - Capturing Business Knowledge with the UML. In *Selected papers from the First International Workshop on The Unified Modeling Language UML'98: Beyond the Notation*, pages 189–199, London, UK, 1999. Springer-Verlag.

[3] Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne Ranta. Multimodal Dialogue System Grammars. In *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue*, pages 53–60, June 2005.

[4] Petra Brosch and Andrea Randak. Position paper: m2n-a tool for translating models to natural language descriptions. *Electronic Communications of the EASST*, Software Modeling in Education at MODELS 2010(34), 2010.

[5] David A. Burke and Kristofer Johannisson. Translating Formal Software Specifications to Natural Language. In Philippe Blache, Edward P. Stabler, Joan Busquets, and Richard Moot, editors, *5th International Conference on Logical Aspects of Computational Linguistics*, volume 3492 of *Lecture Notes in Computer Science*, pages 51–66, Bordeaux, France, April 2005. Springer Verlag.

[6] Jordi Cabot, Raquel Pau, and Ruth Raventós. From UML/OCL to SBVR specifications: A challenging transformation. *Inf. Syst.*, 35(4):417–440, 2010.

[7] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA, 1998.

[8] Donald Firesmith. Modern Requirements Specification. *Journal of Object Technology*, 2(2):53–64, 2003.

[9] Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An Authoring Tool for Informal and Formal Requirements Specifications. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE 2002, Fundamental Approaches to Software Engineering, 5th International Conference*, volume 2306 of *Lecture Notes in Computer Science*, pages 233–248, Grenoble, France, April 2002. Springer.

[10] Christian F. J. Lange, Bart Du Bois, Michel R. V. Chaudron, and Serge Demeyer. An experimental investigation of UML modeling conventions. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2006.

[11] Staffan Larsson and Jessica Villing. The dico project: A multimodal menu-based in-vehicle dialogue system. In *Proceedings of the 7th International Workshop on Computational Semantics (IWCS-7), Tilburg, The Netherlands*. IWCS, 2007.

[12] Peter Ljunglöf. Editing Syntax Trees on the Surface. In *Nodalida'11: 18th Nordic Conference of Computational Linguistics*, volume 11, Riga, Latvia, 2011. NEALT Proceedings Series.

[13] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[14] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[15] Farid Meziane, Nikos Athanasakis, and Sophia Ananiadou. Generating Natural Language Specifications from UML Class Diagrams. *Requirements Engineering*, 13(1):1–18, 2008.

[16] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003.

[17] Peter Naur. Programming as theory building. *Microprocessing and Microprogramming*, 15(5):253 – 261, 1985.

[18] Joaquín Nicolás and José Ambrosio Toval Álvarez. On the generation of requirements specifications from software engineering models: A systematic literature review. *Information & Software Technology*, 51(9):1291–1307, 2009.

[19] Object Management Group. *Semantics of Business Vocabulary and Rules (SBVR) Version 1.0*, formal/08-01-02 edition, January 2008.

[20] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17:40–52, October 1992.

[21] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML^{TM}*. Cambridge University Press, New York, NY, USA, 2004.

[22] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011.

[23] Ehud Reiter and Robert Dale. Building applied natural language generation systems. *Natural Language Engineering*, 3:57–87, March 1997.

[24] Sally Shlaer and Stephen J. Mellor. *Object lifecycles: modeling the world in states*. Yourdon Press, Upper Saddle River, NJ, USA, 1992.

[25] Toni Siljamäki and Staffan Andersson. Performance Benchmarking of real time critical function using BridgePoint xtUML. In *NW-MoDE'08: Nordic Workshop on Model Driven Engineering*, Reykjavik, Iceland, August 2008.

[26] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[27] Leon Starr. How to Build Articulate UML Class Models. http://knol.google.com/k/leon-starr/how-to-build-articulate-uml-class-models/2hnjef6cmm97l/4. Accessed 24th November 2009.

[28] Leon Starr. *Executable UML: How to Build Class Models*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[29] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

Paper 4: Translating Platform-Independent Code

Published as:
Translating Platform-Independent Code into Natural Language Texts
Håkan Burden[1] and Rogardt Heldal[1]
[1] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
MODELSWARD 2013
1st International Conference on Model-Driven Engineering and Software Development
Barcelona, Spain
19-21 February 2013

1 Introduction

In MDA the platform-independent model, PIM, should be a bridge between the specifications in the computationally-independent model, CIM, and the platform-specific model, PSM [19, 21]. Thus it is important that the PIM is clear and articulate [14] to convey the intentions and motivations in the CIM as well as correctly describe the PSM [24].

Since the PSM can be automatically generated from the PIM all changes to the software can be done at PIM-level or on the transformations. In this way the PIM and the PSM are in synchronisation with each other. To keep the CIM and the PIM synchronised is not as easy since their are no automatic transformations from CIM to PIM, yet. Here the translation of the PIM into textual representations can serve as a means of validation of the PIM, in regard to the CIM, during development or to make it easier for new developers to comprehend the structure and behaviour of the system [3].

Claims have been made that comprehensibility is more important than completeness if models are used for communication between stakeholders [22]. But if the stakeholders want to know if the PIM is correct with regards to the software specifications, completeness is just as important. Understanding the annotation and testing of a model requires an understanding of object-oriented design, knowledge of the used models and experience of using the modelling tools [3]. Natural language on the other hand is suitable for stakeholders without the necessary expertise in models and tools [31].

Contributions

This paper shows i) how a platform-independent Action language can be translated into natural language texts ii) by putting natural language generation of software behaviour within the perspective of model-driven software development iii) with transformation rules that are reusable across domains and platforms.

Overview

Section 2 presents the theoretical framework for the study. The tools, technologies and transformations that are used in the study are explained together with examples of translations in section 3. The study is then put in a more general context in the discussion, section 4, before the conclusion is given in section 5. Finally, possibilities to further explore the results are presented in section 6.

2 Theoretical Framework

2.1 Natural Language Generation

Natural Language Generation (NLG; [29]) is a theoretical framework for describing the transformation from software internal models of information into natural language representations. The content, its layout and the internal order of the generated text

is dependent on who the reader is, the purpose of the text and by which means it is displayed. Traditionally NLG is broken down into a three-stage pipeline; *text planning*, *sentence planning* and *linguistic realisation* [29].

Text Planning Text planning is to decide on what information in the original model to communicate to the readers.

Sentence Planning The second stage defines the structure of the individual sentences. This is also the time for choosing the terms that are going to be used for the different concepts. The original software model has now been transformed into an intermediate linguistic model, a grammar.

Linguistic Realisation In the last stage the linguistic model is used to generate text with correct word order and word forms. Through the linguistic realisation the intermediate model has been transformed into natural language text.

2.2 Related Work

Nicolás and Toval [23] provide a systematic literature review on the textual generation from software models. This is a good starting point for a broader investigation into the topic. In their study there is no evidence of text generation from platform-independent Action languages that specify software behaviour.

Recently there has been a flourish of publications on generating natural language from source code. Rastkar et. al. [28] generate English for crosscutting concerns, functionality that is defined in multiple modules, from Java code. As a result of the scattered nature of the crosscutting concerns they are difficult to handle during software evolution. Having a natural language summary for each part of the concern and where it is implemented helps developers handle software change tasks. Sridhara et. al. [32, 33] have also investigated natural language generation from Java code. Their motivation is that understanding code is a time consuming activity and accurate descriptions can both summarise the algorithmic behaviour of the code and reduce the amount of code a developer needs to read for comprehension. The automatic generation of summaries from code mean that it is easy to keep descriptions and system synchronized. An example of a translation from Java to English is found in Figure 20, taken from [32]. Another approach to textual summarisations of Java code is given by Haiduc et. al. [10]. They claim that developers spend more time reading and navigating code than actually writing it. Central to these publications is that they have to have some technique for filtering out the non-functional properties from the source code before translation into natural language.

There are also contributions on using grammars to translate platform-independent specifications into natural language. One such attempt is the translation between the Object Control Language (OCL; [36]) and English [5, 9]. This work was followed up by a study on natural language generation of platform-independent contracts on system operations [11], where the contracts were defined as OCL constraints and specified the pre- and post-conditions of system operations.

Java statement:

```
if (saveAuctions())
```

English translation:

/* If save auctions succeeds */

Figure 20: Example translation of Java to English

3 Exploratory Case Study

In order to explore how a platform-independent Action language can be translated into natural language texts *Executable and Translatable UML* is used to encode the PIM and define the transformation rules. Instead of generating text straight from the PIM the *Grammatical Framework* works as an intermediate modelling language to handle the linguistic properties of the text. In this way the MDA process is integrated with the process of natural language generation.

3.1 Executable and Translatable UML

Executable and Translatable UML (xtUML; [18, 34]) evolved from merging the Schlaer-Mellor methodology [30] with the UML[13] and is a graphical programming language for encoding platform-independent models. BridgePoint[14] was chosen as the xtUML tool.

Three kinds of diagrams are used for the graphical modeling together with a textual Action language. The diagrams are *component diagrams*, *class diagrams* and *state-machines*. There is a clear hierarchical structure between the different diagrams; state-machines are only found within classes, classes are only found within components. Action language can be used in all three component types to define their functional behaviour. The diagrams and action language will be further explained using simplified examples taken from the problem domain chosen for the proof-of-concept implementation, a hotel reservation system.

3.1.1 Diagrams

The xtUML component diagram follows the definition given by UML. In Fig. 21 there is an example of a component diagram. It consists of two components, Hotel and User, connected across an interface.

Fig. 22 shows the class diagram that resides within the Hotel component in the component diagram. The xtUML classes and associations are more restricted than in UML. Only those differences that are interesting for the case study are mentioned. In UML the associations between classes can be given a descriptive association name while in xtUML the association names are automatically given names on the form RN

[13] http://www.uml.org/

[14] http://www.mentor.com/products/sm/model_development/bridgepoint/

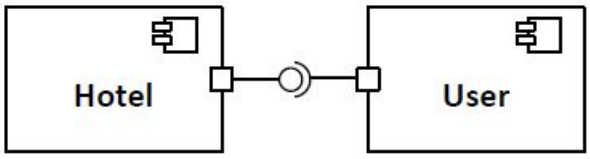

Figure 21: An xtUML component diagram

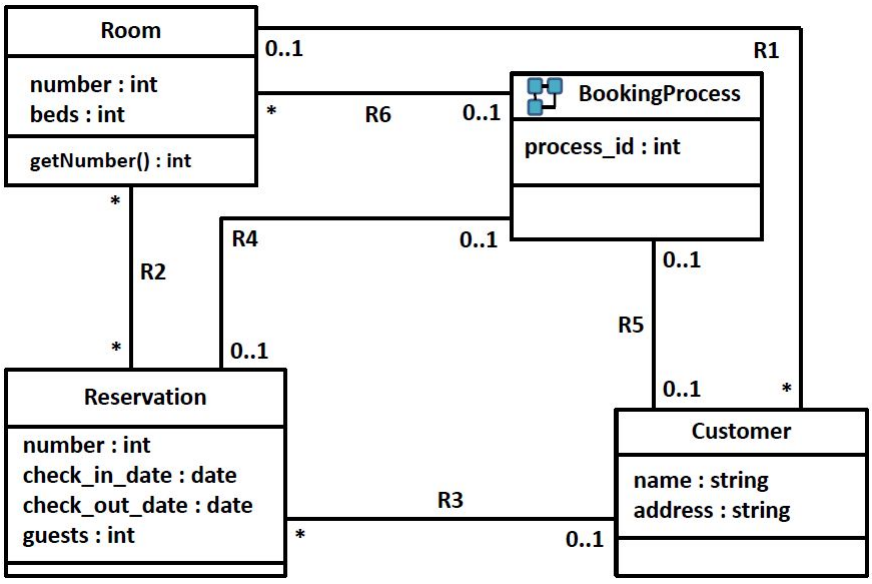

Figure 22: An xtUML class-diagram

where N is a unique natural number. In Fig. 22 Room is associated to Reservation over the association R2. The BookingProcess has no operations, instead the dynamic behaviour is defined by the statemachine residing within, marked by the icon in the top-left corner of the BookingProcess class.

In xtUML a statemachine comprises states, events, transitions and procedures [18]. Fig. 23 shows the statemachine that describes the lifecycles of individual instances of a BookingProcess. Given the statemachine there are two possible transitions from the state Searching; either the event add_room is triggered and the BookingProcess transits to the Adding rooms state or cancel is triggered and the new state is Canceling. If another event is triggered while a BookingProcess is in the Searching state, the event is either ignored or an error is thrown. The states can contain procedures, both events and procedures are defined by the Action language.

3.1.2 Action Language

An important property of xtUML is the Action language. It is a textual programming language that is integrated with the graphical models, sharing the same meta-model [30]. Since the Action language shares the same metamodel as the graphical models it can be used to define how values and class instance are manipulated [15] as well as how the classes change their state [30]. Action language can be used to define the calls between the components as described by the interfaces or to control the flow of calls through the ports of the components. An example of how the Action language can be used is given in Fig. 26. The code details a simple algorithm for finding available rooms and resides within the Searching state of Fig. 23. The example will be further explained in section 3.

The number of syntactical constructs is deliberately kept small. The reason is that each construction in the Action Language shall be easy to translate to any programming language enabling the PIM to be reused for different PSMs. Over the years a number of different Action languages have been implemented [18] and in 2010 OMG released there own standard, ALF[15].

3.1.3 Translating the Models

The xtUML model can be translated into a Platform-Specific Model by a model compiler. A model compiler traverses the metamodel of the PIM and maps each concept into the corresponding concepts of the target language, while preserving the structure of the PIM. Since the platform-specific code is generated from the model, it is possible for the code and the models to always be in synchronization with each other since all updates and changes to the system are done at the PIM-level, never by touching the code.

3.2 Grammatical Framework

Grammatical Framework (GF[16]; [27]) is a domain-specific language for defining Turing complete grammars [6].

3.2.1 GF Grammars

GF separates the grammars into abstract and concrete syntaxes [17]. The abstract syntax is defined by two finite sets, categories (`cat`) and functions (`fun`). The categories are used as building blocks and define the arguments and return values of the functions. From an NLG view the categories are the content and the functions the structure of the text. In the concrete syntax each category and function is given a linearisation definition (`lincat` and `lin` respectively). These definitions gives the sentences their structure and the terminology to be used for the concepts.

A small example of a GF grammar is given in Fig. 24. In the concrete syntax the linearisation of expressions is defined as strings. Integers are represented by their string values which are obtained by record selection, `i.s` [27]. The linearisation rule

[15] `http://www.omg.org/spec/ALF/`
[16] `http://www.grammaticalframework.org/`

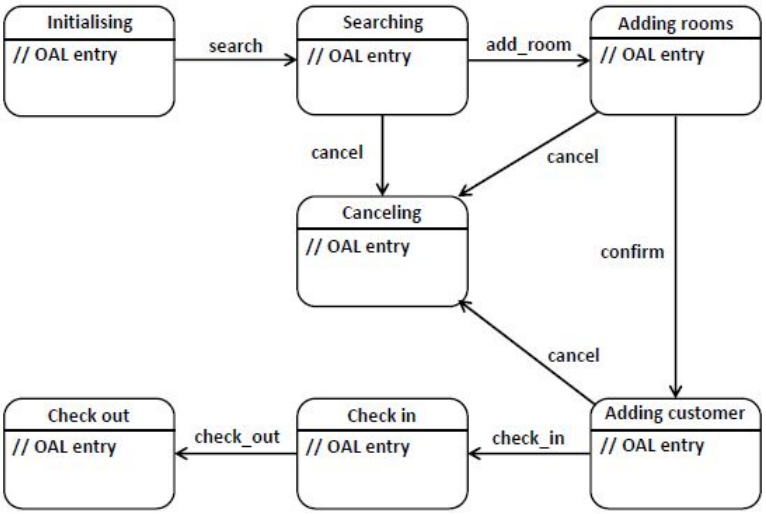

Figure 23: An xtUML statemachine

The abstract syntax:

```
cat Exp
fun Sum  : Exp × Exp → Exp
    EInt : Int → Exp
```

The concrete syntax:

```
lincat Exp = Str
lin    Sum n m = "the sum of" ++ n ++
                 "and" ++ m
       EInt  i = i.s
```

Figure 24: A small GF grammar

for `Sum` is then defined by concatenating the string arguments into their corresponding slots.

An abstract syntax tree defines in which order the functions of the abstract syntax are to be used. A text with multiple readings is ambiguous and will return an abstract tree for each possible reading but each tree will only return one text.

Given the example above the sentence *the sum of 3 and 5* will have the tree

```
Sum (EInt 3) (EInt 5)
```

The transformation from abstract tree to text is called linearisation. Linearisation corresponds to the linguistic realisation of NLG. This transformation is a built-in property of GF [1, 16].

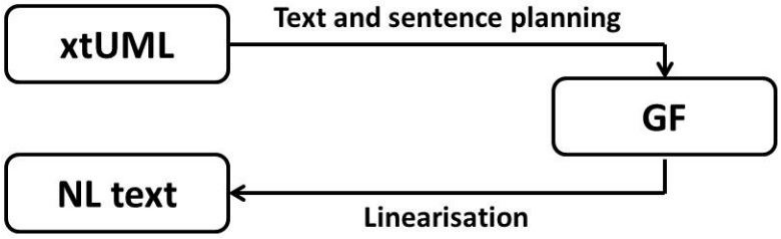

Figure 25: From platform-independent models to natural language texts

3.2.2 The GF Resource Library

In the Resource Grammar Library (RGL; [26]) a common abstract syntax has 24 different implementations in form of concrete syntaxes. Among the concrete languages are English, Catalan and Japanese. The resource grammars have a shared interface which hides the complexity of each concrete language behind abstract function calls. Just as a programmer can use a Java API without knowing how the methods are implemented, the resource grammars support grammar development through an interface that specifies how grammatical structures can be developed [25]. The implementation of each function can be retrieved from the source code and its documentation.

3.3 Model-to-Text Transformations

The automatic translation from software models to natural language texts consists of two transformations, see Fig. 25. The first transformation takes the software model and reshapes it to an intermediate linguistic model by performing text and sentence planning. The second transformation is the linguistic realisation when the linguistic model is used to generate natural language text.

Both transformations are examples of uni-directional and automatic transformations [35]. The first transformation is a reverse engineering translation since the level of abstraction is higher in the target models than in the source models and the two models are defined by different metamodels [20].

Each transformation consists of a set of rules [13] and an algorithm for how to apply the rules [19]. Since the rules of both transformations are defined according to their respective meta-models they are reusable for all models that conform to the same meta-model [4, 19]. The transformations can even be applied to partial xtUML models, enabling textual feedback throughout development on all changes and updates, even if the models need further refining.

3.4 Defining the Grammar

The abstract grammar of the Action language specifies two main categories, expressions and statements. Expressions can be of two kinds, *sentences* or *noun phrases*.

3.4.1 Expressions

From a linguistic point of view a sentence, abbreviated as S, expresses a proposition about the world it inhabits. An example from the Action language is `x == y`, represented in English as *x equals y*. The proposition itself does not claim to be true or false, that is dependent on the context of its evaluation. A characteristic of English propositions are that they follow the form subject-predicate-object, in the example above *x* is the subject, *equals* is the predicate and *y* is the object.

In natural languages, both subjects and objects can have more complicated structures, an example being *the sum of n and m*, written `n + m` in Action language. Such a structure is referred to as a noun phrase, abbreviated as NP. The result of combining the two examples is the expression `x == n + m`, translated as *x equals the sum of n and m*. (Expressions such as `x == y == n + m` can not be formed since the expressions on either side of the equality sign have to refer to members of the program. From a linguistic point of view the expressions have to be NPs.)

This distinction between expressions as sentences and noun phrases is captured in the abstract grammar by the two categories `SExpr` and `NPExpr`. The abstract syntax for the equality function then becomes

```
equality : NPExpr × NPExpr → SExpr
```

with the concrete syntax for English defined using the resource grammars

```
equality x y = mkS (pred (mkV2 "equal" x y))
```

The function `mkV2` takes a string value and returns a verb that expects two NPs, a subject (`x`) and an object (`y`). The function `pred` then takes the verb and the two NPs in order to return an intermediate structure that is passed on to `mkS`. The result of applying `mkS` is a sentence on the form *x equals y* where both *x* and *y* can be complex NPs. In order to handle agreement between subject and verb the linearisation categories for nouns and verbs have to be more complex than just strings. Exactly how complicated is not a problem for those using the RGL as an API for grammar development, it has already been dealt with by the RGL developers. Instead, the complexity lies in applying the appropriate functions from the API in the right order.

Both the S- and NP-expressions are derived from the xtUML metamodel where they are encoded as subtypes of the metaclass Value or as instances of Variable. In the above example for `equality` both the binary operation and the `NPExpr` are defined as Values. By recursively analysing the left and right expressions of the operation shows that `x` and `y` are instances of the metaclass Variable with their respective names. Unary operations, attribute references and parameters for events and operations are other subclasses of Value.

3.4.2 Statements

If expressions could be both noun phrases and sentences, all statements are sentences. An example of this is the Action language's return statement `return x` where `x` could be both an NP such as *the sum of n and m* as well as a sentence, *n equals m*. The solution is to have two abstract functions defining the return statement, one for returning noun phrases and one for returning sentences

```
returnNP : NPExpr → Stmt
returnS  : SExpr  → Stmt
```

For the concrete syntax a more general phrasing than *return n* is used since it can be unclear for non-programmers to whom *n* is returned and what this means. This decision highlights how the abstract syntax defines the text planning of the natural language generation while the concrete syntax defines the words to be used for different concepts and how these words are to be strung together, i.e. the sentence planning.

The first function for return statements is implemented in a fashion similar to the one used for equality expressions

```
returnNP n =
  mkS (pred n (mkNP the_Det (mkN "result")))
```

and returns statements such as *the result is the sum of n and m* for `return n + m`. For returning sentences other functions from the RGL are used since the type of the argument is different

```
returnS  s =
  mkS (mkCl (mkNP the_Det (mkN "result")) s)
```

As an example *the result is x equals y* is the equivalent translation for `return x == y`.

Finally, a program is defined as a list of statements

```
fun sequence : [Stmt] → Prgm
```

3.5 Translations

The diagram in Fig. 26 shows an example of a program written in Action language side-by-side with its translation where the Action code resides within the Searching state shown in Fig. 23. The generated text is an example of a controlled natural language (CNL; [37]) where the described language is a subset of a natural language. A common aspect of such languages is that they are perceived as lacking in naturalness [7] and that the sentences have a repetitive structure inherited from the source model. This can also be a benefit since it allows readers to quickly recognise and interpret the different sentence structures [7].

The Action language is platform-independent in the sense that it makes no assumptions on how collections are to be implemented, all collections are treated as sets. This is exemplified on line 6 where many `Room`s are selected and stored as a set using the variable `rooms`. On line 7 a for-loop is used to iterate over the set. On the other side, the Action language is not independent from the object-oriented modelling paradigm. This shows in lines 1 and 2 where an instance of an object is created and then associated to another object. To interpret the Action language requires an understanding of the implicit information encoded in the paradigm of object-oriented languages [3]. The aim of the translation is to make such information explicit without being to lengthy. Another aspect of the underlying design choices of the Action language is shown in the naming convention for traversing across associations. Here the unique association

```
create object instance res of Reservation;      res refers to a Reservation
relate res to self across R4;                   res and the BookingProcess
                                                    share information
res.check_in = param.in;                        res's check in gets the value
                                                    of the given in
res.check_out = param.out;                      res's check out gets the value
                                                    of the given out
res.guests = param.quantity;                    res's guests gets the value
                                                    of the given quantity
room_number = 0;                                room number gets the value of 0
select many rooms from instances of Room;       rooms refers to many Rooms

for each room in rooms                          for each room in rooms
  relate self to room across R6;                the BookingProcess and room
                                                    share information
  select many ress related by                   ress refers to many Reservations
    room -> Reservation[R2];

  for each res in ress                          for each res in ress
    if (res.check_in > param.out                if res's check in is greater
                                                    than the given out
     or res.check_out < param.in)               or res's check out is less than
                                                    the given in
     and room.beds == param.quantity            and room's beds equals the given
                                                    quantity
      room_number = room.getNumber();           then room number gets the value
                                                    of room's get Number
      break;                                    the for-loop is terminated
    end if;
  end for;

  if room_number > 0                            if room number is greater than 0
    break;                                      the for-loop is terminated
  end if;
end for;

if room_number == 0                             if room number equals 0,
  send HotelInterfaces::                            then a cancellation with
    cancellation(process_id:self.process_id,        process id and message
      message:"No available rooms.");               is sent to User
else                                            else a confirm room with
  send HotelInterfaces::                            process id and room
    confirm_room(process_id:self.process_id,        is sent to User
      room:room_number);
end if;
```

Figure 26: An example of Action language code with natural language summarisation

names are used, which have no relevance for the domain. In the translation to natural language texts association names, such as `R2`, are therefor not mentioned.

Just as graphical models the Action language is supposed to deliver a high-level view of the system. But the abstraction gets muddled by language-specific details such as the association names and the object-oriented syntax, concepts that are not meaningful to all stakeholders [8].

The generated text is dependent on that meaningful values have been assigned to class names, parameters etc. If the class Reservation was named RSV instead the translation would generate sentences such as *res refers to an RSV* making the generated texts harder to comprehend.

On line 2 the statement `relate res to self across R4` could have been translated as *relate res to self*. But what does it mean that two objects are related? From an object-oriented view it means that they can access each other's public attributes and operations. The translation tries to capture this without going into details about the fundamentals of object-oriented design, substituting the reference `self` for the definite form of the class name of the referent, *the BookingProcess*.

The Action code finishes by sending a signal across the interface to the User component. Depending on if a room was found or not different signals are sent. Here the name of the interface, `HotelInterfaces` is substituted for the more informative `User` which is found by traversing the metamodel across the interface and its ports to the receiving component.

The signals exemplify a challenge for generating summarisations; should the parameters be translated using the parameter name, its defining expression or both? In the case of the `message` the expression is more descriptive than the name but for the `room:room_number` parameter both name and expression would be useful. The value of the `process_id` is less informative than the parameter name (`process_id` is included as a parameter to ensure that the right instance of `BookingProcess` gets the reply from the User). To make an informed decision on the best phrasing in each case would require a semantic analysis of the values of the parameter expressions in comparison to the parameter names, something that is not supported by the transformation language.

4 Discussion

4.1 Changing the Language

Different stakeholders have different needs in terms of the content of the summarisations, e.g. the developers want a quick introduction to the functionality of the system [32] while domain experts want to validate that certain requirements are met and maintained [3]. This can be accommodated by using different transformation rules for generating the grammars. One transformation can then generate a grammar that produces summarisations for the developers while another transformation is aimed towards the needs of the domain experts. The result is a shared abstract syntax that is realised by different concrete syntaxes to fit their respective needs using different functions from the RGL.

Some stakeholders might prefer another language than English. This can be facilitated by the multilingual aspect of the Grammatical Framework. In this approach the lexicon (or domain vocabulary) of the grammar is generated from the Action language. However, it is not obvious that the domain concepts share their names across languages. There are two ways to overcome this challenge; The naïve way is to ensure that the modelling elements use the terminology of the desired target language, by this approach the lexicon is automatically generated in the desired language. The other solution is to manually develop a lexicon per desired language, as explained in [2]. Since the abstract functions defined by the RGL are language-independent, the same rules can be used for all desired languages. In this way the structure and content of the texts are preserved but with language-specific implementations of the sentences.

It is important to remember that any changes to the grammars are made through the transformation rules. As a consequence the transformation experts need to know the grammar that is used to model the texts well enough to implement the changes. It also means that neither the software modellers nor the customers need to know how the text is generated or how to formulate model transformations. When the transformations have been defined the translations are generated by a push on the button. The generation can then be repeated and reused for all models that conform to the same metamodel as the transformation rules [4, 19].

4.2 The Complexity of Model Transformation

The complexity of the model transformations does not lie in the complexity of the transformation rules but in the complexity of the modelling language they are applied to [12].

On the target end of the transformation a knowledge of linguistics in general and the grammar API is needed to utilise the different categories and functions of the grammar in an efficient way. The alternative to grammars would be to generate text straight from the models with the tedious work of making sure that there is congruence between the verbs and the noun phrases as well as taking care of aspects like *a reservation* but *an interface*.

4.3 Text vs Models

Another benefit of natural language translations of textual software models embedded in graphical model elements is that they enable using any preferred text editor for searching after concepts and actions that should be in the text. Different modelling tools have their own support for searching with different interfaces, learning how to use them all is a tall request on stakeholders [3].

5 Conclusions

The proposed way of translating Action code differs from previous work on code summarisation in that the platform-independent models already have filtered away the non-functional properties of the software, leaving the functional properties exposed.

In comparison to previous research on generating natural language texts from software models this is the first attempt to generate software behaviour from platform-independent code.

The PIM can be reused to generate a number of different platform-specific models that include the usage of different APIs, programming languages, connections to operative systems and deployment on hardware. Since, the functionality of the system is captured in the PIM so the generated text gives a natural language summary of the system's behaviour disregarding how this behaviour is implemented. This means that the generated text can be used across platforms and updated by re-generation whenever the PIM is changed to reflect new requirements or bug-fixing. So, instead of having one framework for translating Java, another framework for translating C and a third for C++, a general framework for translating platform-independent code can be reused across platforms independently of how the system is realised.

6 Future Work

The mapping rules that define the transformation from PIM to PSM add the non-functional features that determine a certain combination of platform-specific details. Generated summarisations from the mappings could then describe the different profiles and properties of the system, such as safety and persistency.

The challenges in natural language generation from the combination of textual and graphical models is an interesting step to further explore. A case study is planned for including transformation rules that map the structure of the statemachines on to the generated translations. In this way the translations will give an overall structure of the software that follows the lifecycles of the system's classes and objects.

Bibliography

[1] Krasimir Angelov. *The Mechanics of the Grammatical Framework*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2011.

[2] Krasimir Angelov and Aarne Ranta. Implementing Controlled Languages in GF. In *Controlled Natural Language, Workshop on Controlled Natural Language, CNL 2009, Marettimo Island, Italy, June 8-10, 2009*, volume 5972 of *Lecture Notes in Computer Science*, pages 82–101. Springer Verlag, 2009.

[3] Jim Arlow, Wolfgang Emmerich, and John Quinn. Literate Modelling - Capturing Business Knowledge with the UML. In *Selected papers from the First International Workshop on The Unified Modeling Language UML'98: Beyond the Notation*, pages 189–199, London, UK, 1999. Springer-Verlag.

[4] Colin Atkinson and Thomas Kühne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36 – 41, September 2003.

[5] David A. Burke and Kristofer Johannisson. Translating Formal Software Specifications to Natural Language. In Philippe Blache, Edward P. Stabler, Joan Busquets, and Richard Moot, editors, *5th International Conference on Logical Aspects of Computational Linguistics*, volume 3492 of *Lecture Notes in Computer Science*, pages 51–66, Bordeaux, France, April 2005. Springer Verlag.

[6] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[7] Peter Clark, William R. Murray, Philip Harrison, and John A. Thompson. Naturalness vs. Predictability: A Key Debate in Controlled Languages. In *Controlled Natural Language, Workshop on Controlled Natural Language, CNL 2009, Marettimo Island, Italy, June 8-10, 2009*, volume 5972 of *Lecture Notes in Computer Science*, pages 65–81. Springer Verlag, 2009.

[8] Andrew Forward and Timothy C. Lethbridge. Problems and Opportunities for Model-Centric Versus Code-Centric Software Development: A Survey of Software Professionals. In *Proceedings of the 2008 international workshop on Models in Software Engineering*, MiSE '08, pages 27–32, New York, NY, USA, 2008. ACM.

[9] Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An Authoring Tool for Informal and Formal Requirements Specifications. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE 2002, Fundamental Approaches to Software Engineering, 5th International Conference*, volume 2306 of *Lecture Notes in Computer Science*, pages 233–248, Grenoble, France, April 2002. Springer.

[10] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In Giuliano Antoniol, Martin Pinzger, and Elliot J. Chikofsky, editors, *WCRE*, pages 35–44. IEEE Computer Society, 2010.

[11] Rogardt Heldal and Kristofer Johannisson. Customer Validation of Formal Contracts. In *OCL for (Meta-)Models in Multiple Application Domains*, pages 13–25, Genova, Italy, 2006.

[12] Jean-Marc Jézéquel, Benoît Combemale, Steven Derrien, Clément Guy, and Sanjay Rajopadhye. Bridging the Chasm Between MDE and the World of Compilation. *Journal of Software and Systems Modeling (SoSyM)*, pages 1–17, 2012.

[13] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture^{TM}: Practice and Promise*. Addison-Wesley Professional, Boston, MA, USA, 2005.

[14] Christian F. J. Lange, Bart Du Bois, Michel R. V. Chaudron, and Serge Demeyer. An experimental investigation of UML modeling conventions. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2006.

[15] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[16] Peter Ljunglöf. Editing Syntax Trees on the Surface. In *Nodalida'11: 18th Nordic Conference of Computational Linguistics*, volume 11, Riga, Latvia, 2011. NEALT Proceedings Series.

[17] J. Mccarthy. Towards a Mathematical Science of Computation. In *Proceedings of the Information Processing Congress*, pages 21–28. North-Holland, 1962.

[18] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[19] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[20] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.

[21] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003.

[22] Parastoo Mohagheghi and Jan Aagedal. Evaluating quality in model-driven engineering. In *MISE '07: Proceedings of the International Workshop on Modeling in Software Engineering*, page 6, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Joaquín Nicolás and José Ambrosio Toval Álvarez. On the generation of requirements specifications from software engineering models: A systematic literature review. *Information & Software Technology*, 51(9):1291–1307, 2009.

[24] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17:40–52, October 1992.

[25] Aarne Ranta. Grammars as software libraries. In G. Huet, G. Plotkin, J.-J. Lévy, and Y. Bertot, editors, *From semantics to computer science: essays in honor of Gilles Kahn*. Cambridge University Press, 2008.

[26] Aarne Ranta. The GF Grammar Resource Library. *Linguistic Issues in Language Technology*, 2(2), 2009.

[27] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011.

[28] Sarah Rastkar, Gail C. Murphy, and Alexander W. J. Bradley. Generating natural language summaries for crosscutting source code concerns. In *27th International Conference on Software Maintenance*, pages 103–112, Williamsburg, VA, USA, September 2011. IEEE.

[29] Ehud Reiter and Robert Dale. Building applied natural language generation systems. *Natural Language Engineering*, 3:57–87, March 1997.

[30] Sally Shlaer and Stephen J. Mellor. *Object lifecycles: modeling the world in states*. Yourdon Press, Upper Saddle River, NJ, USA, 1992.

[31] Silvie Spreeuwenberg, Jeroen Van Grondelle, Ronald Heller, and Gartjan Grijzen. Design of a CNL to Involve Domain Experts in Modelling. In Michael Rosner and Norbert Fuchs, editors, *CNL 2010 Second Workshop on Controlled Natural Languages*, pages 175–193. Springer, 2010.

[32] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[33] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 101–110, New York, NY, USA, 2011. ACM.

[34] Leon Starr. *Executable UML: How to Build Class Models*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[35] Perdita Stevens. A Landscape of Bidirectional Model Transformations. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424, Braga, Portugal, July 2007. Springer.

[36] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[37] Adam Wyner, Krasimir Angelov, Guntis Barzdins, Danica Damljanovic, Brian Davis, Norbert Fuchs, Stefan Hoefler, Ken Jones, Kaarel Kaljurand, Tobias Kuhn, Martin Luts, Jonathan Pool, Mike Rosner, Rolf Schwitter, and John Sowa. On controlled natural languages: Properties and prospects. In Norbert E. Fuchs, editor, *Proceedings of the Workshop on Controlled Natural Language (CNL 2009)*, volume 5972 of *Lecture Notes in Computer Science*, pages 281–289, Berlin / Heidelberg, Germany, 2010. Springer Verlag.

Appendix C:
Scholarship of Application

Paper 5:
Multi-Paradigmatic Modelling of Signal Processing

Published as:
Industrial Experiences from Multi-Paradigmatic Modelling of Signal Processing
Håkan Burden[1], Rogardt Heldal[1] and Martin Lundqvist[2]
[1] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
[2] Baseband Research
Ericsson AB
MPM'12
6th International Workshop on Multi-Paradigm Modeling
Innsbruck, Austria
1 October 2012.

1 Introduction

Embedded software applications in industry are often composed of an interacting set of solutions to problems originating from different domains. Although these problem domains may be naturally separated, and initially specified by different expert designers using different appropriate methods, the actual implementation of the combined application is often delegated to programmers using general program languages. The programmers are forced to add program language-dependent details in their manually produced code, including optimizations for the current hardware. The result is a mix-up of the desired functionality and structure of the system together with the hardware-specific details, all intertwined in the syntax of the program languages used for implementation.

We decided to explore the possibilities for using multiple modelling languages in implementing the channel estimation of the LTE-A uplink test bed of a 4G telecommunication system [8]. The requirements on such an application includes unconditional real-time performance for calculations on synchronous data and the contextual determination of when signals shall be sent and processed as well as operation reliability.

In our contribution we show that it is possible to approach such a system by identifying its different domains - based on their main properties and respective needs for expression - and the interfaces between them. Each domain can then be implemented independently by using an executable and translatable modelling language.

The related work is presented in section 7, followed by our motivation in section 2. In section 3 we give the necessary background about the investigated domains and what we consider suitable modelling languages for each domain. In section 5 the delivered application is described together with the implementation process. The outcome of the process is then found in section 6 which is followed by our discussion, section 7. Finally, we conclude and propose new questions for further research in section 8.

2 Related work

A substantial part of research has been reported for combining multiple modelling or domain-specific languages from the perspective of meta-modelling (e.g. see [5, 16, 20, 24, 27, 29]).

Another approach to multi-paradigmatic modelling is reported by Lochmann and Hessellund [18]. They give a schema for working with multiple modelling languages where the first step is to identify the different domains and their connections. Then the connections are specified and finally the domains and the connections are implemented. In this way the different languages do not need to be combined on the level of metamodels. A challenge for Lochmann and Hessellund is to ensure referential integrity across the connections and domains. Hessellund *et al.* [15] have developed a tool that was used for this purpose in an industrial project, SmartEMF. A similar approach is taken by Nentwich *et al.* [21], Denton *et al.* [9] and Warmer and Kleppe [30].

Motorola has applied model-driven engineering to describe the asynchronous mes-

sage passing in a telecommunication system [7]. They discuss the challenges when different views of the system are integrated and how these can be overcome by using an aspect-oriented approach [11]. The Motorola authors implemented the signal processing by hand-written code.

3 Motivation

In order to implement the LTE-A uplink channel estimator we decided to use two different modelling languages; a functional language to model the algorithms of the signal processing and an object-oriented language to model the execution flow of the signals. The different paradigms of the modelling languages will, ideally, ensure that the platform-independent models are closer to the requirements and domain descriptions than in the case of using one general modelling language [20]. In this way the functionality and structure of the domain implementations are to be free of platform-specific intrusions.

The setup poses two challenges:

1. To which extent is it possible to develop the signal processing and the flow of execution independently of each other?

2. How can the sub-solutions be integrated into one application running on the designated hardware?

Before we explore the challenges further we need to further define what we mean by a domain.

4 Background

After defining what we mean by domain we describe the two domains in the LTE-A uplink channel estimator, the signal processing domain and the control domain.

4.1 Domain Definition

In our context a domain represents one subject matter with a set of well-defined concepts and characteristics [25] that cooperate to fulfill the interactions of the domain [23]. This definition of a domain is in line with what Giese *et al.* [12] call a horizontal decomposition and ensures the separation of concerns between the domains [10] as well as information hiding [22]. Furthermore, each domain can be realised as one or more software components [23].

4.2 The Signal Processing Domain

4.2.1 Signal Processing

What is often referred to within Ericsson AB as the signal processing domain, is characterized by a data centralized processing flow, where program state changes and

$$\text{DCT-2}_n = \left[\cos \frac{k(2l+1)\pi}{2n}\right]_{0 \leq k,l < n}$$

```
dct2 :: DVector Float -> DVector Float
dct2 xn = mat ** xn
  where mat =
  indexedMat (length xn) (length xn)
    (\k l -> dct2nkl (length xn) k l)

dct2nkl n k l =
  cos ((k'*(2*l'+1)*3.14)/(2*n'))
    where (n',k',l') = (intToFloat n,
                        intToFloat k,
                        intToFloat l)
```

Figure 27: The Discrete Cosine Transform matrix in mathematical and Feldspar notation.

external interactions are kept at a minimum, while more or less fixed and carefully optimized algorithms filter, convert or otherwise calculate on incoming data in a predeterministic way. In telecommunication applications, signal processing plays a crucial role, and the necessary algorithms have to be efficient in order to achieve the performance required on speed and quality.

In the LTE-A uplink testbed project, the signal processing more precisely consisted of multiple-user, multiple- antenna uplink data processing according to the 3GPP[17] standard, and our modelling mainly implicated the channel estimation parts.

4.2.2 Implementing Signal Processing

Today, using general high level languages, such as C or similar, there is often a large gap between the algorithm design and the implementation of the same. Due to this gap, the implementation of the signal processing algorithms can be indirect and unnecessarily complicated, and therefore error prone.

Feldspar is a domain-specific language currently developed by Chalmers University of Technology and Ericsson for signal processing [2]. The purpose is to limit the gap between the mathematical notation used in the design of signal processing algorithms and their implementation by using a functional modelling paradigm. Feldspar is embedded in Haskell[18], a third generation functional programming language, and tries to remain true to the Haskell syntax [3]. One important aspect of Feldspar is that it has no side-effects.

A Feldspar program can be evaluated directly via a Haskell interpreter, or be transformed into C by the accompanying code generator. In Figure 29 there is an example of a mathematical matrix multiplication used in signal processing together with the equivalent Feldspar definition, taken from Axelsson *et al.* [2].

[17] `http://www.3gpp.org/specifications/`
[18] `http://www.haskell.org/`

4.3 The Control Domain

4.3.1 Controlling the Flow of Execution

We define the term Control Domain as a part of a software application controlling the flow of execution; responding to external communication and perhaps governed by internal state machinery. The control domain itself does not contain any complicated algorithmic complexity, instead it controls the order in which things are executed; in our case receiving and sending signals, initiating signal processing routines, and collecting their results.

4.3.2 Executable and Translatable UML

Our previous experiences at Ericsson show that an object-oriented modelling language is well suited for implementing the solutions needed for the control domain; modelling the interaction with surrounding applications in the system and managing the control and exchange of data between the different parts of the signal processing domain, regardless of implementation language chosen for those parts. Similar experiences from the telecommunications industry are reported on by Weigert and Weil [31].

Executable and Translatable UML (xtUML; [19, 23, 28]) evolved from merging the Shlaer-Mellor method [25] with the Unified Modeling Language (UML[19]).

xtUML has three kinds of diagrams, together with a textual action language. The diagrams are component diagrams, class diagrams and state machines. There is a clear hierarchical structure between the different diagrams; state machines are only found within classes, and classes are only found within components. Component diagrams have more or less the same syntax as in UML, but both class diagrams and state machines are more restricted in their syntax in comparison to UML. The action language is integrated with the graphical diagrams by the shared metamodel. At the time of our project the metamodel was propriety with restrictions on how it could be extended. The number of constructions in the metamodel is deliberately kept small so that there is always an appropriate correspondence in the platform-specific model. This also makes it an unsuitable language for complex algorithms since it has a very limited set of datastructures and only fundamental mathematical notations.

Since xtUML models have unambiguous semantics validation can be performed within the xtUML tool by an interpreter. During execution all changes of the association instances, attribute values and class instances are shown [17], as well as the change of state for classes with state machines, in the object model.

4.4 The Interface between the Domains

The actual processing of incoming data was - however static regarding the contents and order of algorithms involved - completely dynamic depending on the current configuration. In one instant, a certain configuration would put emphasis on a specific algorithm, quickly changing with a new configuration the next millisecond. This called for the need for independent possibilities for parallelization of each of the seven main

[19] `http://www.uml.org/`

algorithms, in order to continuously keep the processing latency at a minimum. This dynamicity in the control of invocation of algorithms decided in great extent the interfaces between the two identified domains.

5 Case Study

5.1 Context

The application chosen for our case study was part of a larger Ericsson testbed project, already involving legacy and new software and hardware, and also new features. The testbed mainly involved 4G telecommunication baseband functionality, based on Ericsson's existing LTE products, both base station parts and user equipment parts. The testbed included adding features as well as deploying applications on a new hardware platform, resulting in an LTE-A [8] prototype to be presented at the Mobile World Congress in Barcelona, February 2011. Due to compiler availability for the new hardware the model transformations were restricted to generate C code.

The testbed project lasted for over one year. Already from the beginning it was emphasized that delivering an application that fulfilled the requirements within the calender deadline was more important than using specific methods and languages. There were no intentions of reusing anything of what the modelling could bring, except perhaps from the experienced gained.

5.2 Domain Identification

The application considered in this project was identified as a self-contained software component, managing a specific part of the data flow in an LTE-A base station, see Figure 28. The component's external interfaces were well specified, both in terms of parameter interchange and real-time responsibilities. The application was to be periodically provided with incoming data, while independently requested to update its configuration regarding how to process the data. Upon external triggering in one of these ways the control domain initiated internal chains of signal processing which were expected to run to completion.

5.2.1 Modelling Signal Processing using Feldspar

It was decided that it would be a good opportunity to test Feldspar by modelling the signal processing. There were other options, such as MatLab[20], but we wanted to take the opportunity to evaluate Feldspar's capability for modelling signal processing in a sharp project.

5.2.2 Modelling Control using xtUML

Ericsson has previously had success in using xtUML for reuse of platform-independent models [1] and test generation [6] while still finding the tool easy enough to use by novice modellers [4]. We also know from experience that xtUML integrates nicely

[20] `http://www.mathworks.se/products/matlab/`

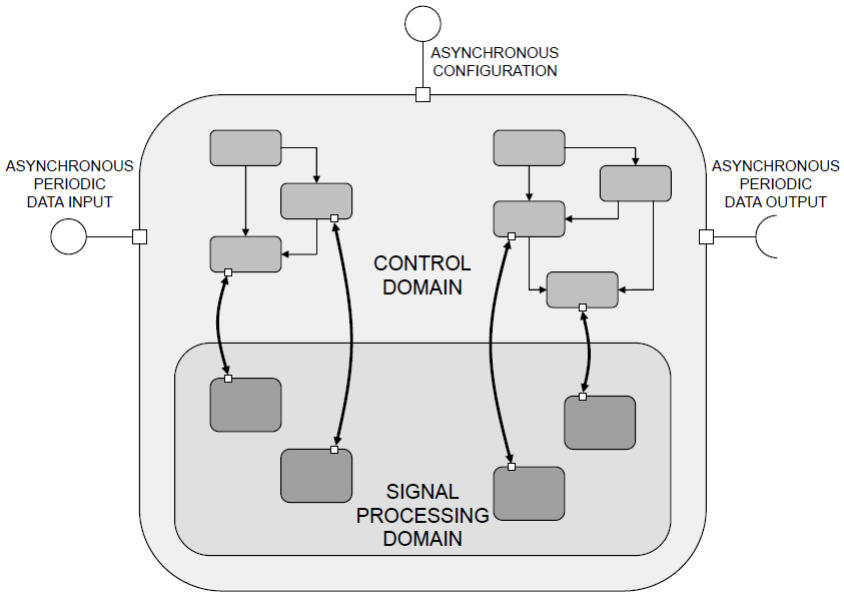

Figure 28: A schematic representation of how the state machines in the control domain cooperate with the independent algorithms of the signal processing domain.

with legacy code written in C. Since the models are executable it is possible for users to validate that the models have the required functionality without generating code for deployment. Ericsson also had an in-house xtUML-to-C transformer that could compete with hand-written code [26]. And it was decided that one new modelling language was enough considering the firm deadline of the project. BridgePoint[21] was chosen as tool for modelling xtUML.

5.2.3 Modelling the Interface between the Domains

The interface between the two domains was defined in an implementation language independent way, identifying parameters necessary for describing the algorithmic and parallelization needs of the different algorithms within the data processing chain. Lochmann and Hessellund refer to such an implementation-independent interface as semantic [18]. The functionality of the interface was analysed using activity diagrams which was then manually transformed into C code.

5.3 Developers

The people involved in the project had been working between 5 to 15 years each with layer one baseband signal processing in telecommunication equipment at Ericsson. Mainly three developers were involved in the implementation of the models, two implementing the signal processing domain using Feldspar and one developer using xtUML

[21] `http://www.mentor.com/products/sm/model_development/bridgepoint/`

for implementing the control domain. The developers had an unusual combination of expertise in that they were both domain experts and proficient C coders. In addition to the implementors, there was one domain expert in designing signal processing algorithms linked to the project as well as two experts in model transformations; one transformation expert for each modelling language. These two also served as mentors in respective modelling language. Two academic researchers participated as observers throughout the whole project.

The domain experts had chosen to participate in the project themselves. They had also chosen the modelling languages to use for implementing their domains. Their knowledge of C was a key reason for attempting to use a new modelling language, if Feldspar did not deliver it would always be possible to revert to the old ways and implement the signals using C. The motivation and commitment of the developers towards the project was one of the reasons that the project delivered a successful application.

5.4 Operation

The operation of the project was done in a Scrum way[22], using a backlog and in each sprint there were daily meetings and a burn down chart. After the end of each sprint there was a sprint review and delivery.

The testbed was horizontally decomposed [12] into a control and a signal processing domain according to the domain identification. The control domain was viewed as one autonomous component while the algorithms were developed as components of their own residing inside the control component. The implementation was done following a component-based development principle [13]. Our choice of language for each domain enabled testing of each software component independently and continuously throughout the implementation. When the implementation was complete the models were transformed into source code through code generation.

6 Results

From the outcome of the case study we can now answer the challenges in section 2 in the order they were specified; first, to which extent is it possible to independently implement the domains? and secondly, how can the sub-solutions be integrated and deployed to constitute a running application?

6.1 Independent Implementation and Validation of the Domains

Working independently in the two modelling languages went well. The Feldspar models for the algorithms in the signal processing domain were implemented independently. Since the signal processing consisted of discretely implemented algorithms, they could be verified independently, processing premade input data, and comparing the output with likewise premade output data using the Haskell interpreter.

[22] `http://www.scrum.org/scrumguides/`

Simultaneously and independently, the xtUML models for the control domain could be implemented and tested. For the places where functionality defined in Feldspar were to be called from xtUML, stubs defined using action language were used within the executable control domain model, in order to verify the complete control flow; albeit not in a complete real-time aspect.

The designers implementing in Feldspar found it suitable for writing mathematical expressions, although it took some time to get used to the Feldspar syntax. There was initially only limited support for fixed-point arithmetics which is an important notion within the domain and the Haskell syntax for representing state was not intuitive compared to the mathematical formulation. The mapping from the mathematical notation of one of the algorithms into Feldspar code was done by the Feldspar mentor since it was not possible for the domain experts to do.

Using xtUML was straight-forward when modelling the solution for the control domain. Activity diagrams would have been preferable, since they would better describe the parallel execution of the signals than the state machines. The solution in our case was to model the control domain basic structure as activity diagrams outside the xtUML tool, in order to understand how to translate the behavior using the state machine diagrams provided by xtUML.

6.2 Integrating the Domain Solutions

Defining the interfaces required iteration over several meetings where the interfaces had to be refined and adapted due to extensions and changes on the functionality in the requirements. Since the understanding of the requirements on the interface increased during the implementation of the separate domains. These refinements were handled through the agile principles[23] inherent in the Scrum development process.

Testing the full interaction between the Feldspar and xtUML models required that they first were transformed into C code. The generated code was then validated before being deployed on the delivered hardware. Due to performance limitations some hand-written C code with added intrinsics was necessary when deploying the generated code to maximise the usage of the platform-specific properties. Under the time constraints of the project this was quicker than updating the existing transformations [14].

7 Discussion

7.1 The Right Language for the Right Task

The Feldspar developers needed time to get used to writing Feldspar code due to the design of the language. The authors of Feldspar claim that it is a domain-specific language for signal processing. Often, when developing a domain-specific language, one starts with the syntax used by the domain experts, which in our case would have been mathematical expressions for signal processing. This was not the case for Feldspar, since it is deliberately defined to have as similar syntax as possible as Haskell. Even though the gap between signal processing algorithms and Feldspar is a lot smaller than

[23] `http://agilemanifesto.org/principles.html`

between signal processing algorithms and C there is still a gap. Moreover, Feldspar was a new language to learn for the domain experts and it belongs to a different programing paradigm than C which they were used to. The problem was limited by the fact that one of the people working in the Feldspar project was part of the group during the initial phase of the project.

From our experience of xtUML [4] it does not take that long time to get used to the syntax of the modelling elements, as it does to get used to the tool that creates them. More importantly, as the project evolved we saw a need for handling the data flow in a way that neither xtUML nor the existing platform supported.

7.2 Multicore

At the time of our project there was no way to generate efficient code for multicore deployment, neither from Feldspar nor from xtUML, so we opted to implement the interface in C straight away. However, both languages chosen for its respective domain suggested suitable inherent properties for such a transformation to be plausible in many aspects. The feature of Feldspar functions having no side effects, together with designing the algorithms as a library of dynamically composable low-level operations would be well suited for execution in a distributed and concurrent manner. And implementing dynamically created independent instances of state machinery in xtUML would only benefit from being run as distributed and concurrent threads in a multicore environment.

Exploiting properties already naturally inherent within the domain languages, when transforming the domain models into generated code, would therefore render it possible to generate code suitable for multicore deployment - for free, so to speak. Naturally it would require considerately more development by the transformation experts to customise the transformations in order to obtain optimal performance for a specific platform.

8 Conclusion and Future work

In relation to previous work we chose not to unify the metamodels of the modelling languages since we did not have straight access to the xtUML metamodel. Instead we relied on the identification of a logical interface between the different domains and modelling languages. In this aspect our work is related to the findings reported by Lochmann and Hessellund [18]. By adhering to an agile and component-based development strategy [13] and through the separation of concerns [10] between the domains, the referential integrity is ensured through the interface between the domains [23]. In comparison to the work at Motorola [7, 31] we have used an in-house modelling language to implement the signal processing.

The development process relied on the possibility to validate the domains independently. For this to be possible it is important that the modelling languages are executable in their own right and that they later on can be translated into deployable code, after the models have been validated to have the right structure and behaviour [23].

We believe that multicore is both a challenge and an opportunity for model-driven software development. Multicore is a complex paradigm of its own and if the platform-independent properties of software modelling could be combined with efficient code generation a lot would be won.

Bibliography

[1] Staffan Andersson and Toni Siljamäki. Proof of Concept - Reuse of PIM, Experience Report. In *SPLST'09 & NW-MODE'09: Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, Tampere, Finland, August 2009.

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169 –178, July 2010.

[3] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar – An Embedded Language for Digital Signal Processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] Håkan Burden, Rogardt Heldal, and Toni Siljamäki. Executable and Translatable UML – How Difficult Can it Be? In *APSEC 2011: 18th Asia-Pacific Software Engineering Conference*, Ho Chi Minh City, Vietnam, December 2011.

[5] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the meta-model level: the Fujaba approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6:203–218, 2004.

[6] Federico Ciccozzi, Antonio Cicchetti, Toni Siljamäki, and Jenis Kavadiya. Automating test cases generation: From xtUML system models to QML test models. In *MOMPES: Model-based Methodologies for Pervasive and Embedded Software*, Antwerpen, Belgium, September 2010.

[7] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Motorola WEAVR: Aspect and Model-Driven Engineering. *Journal of Object Technology*, 6(7):51–88, August 2007. Aspect-Oriented Modeling.

[8] E. Dahlman, S. Parkvall, and J. Sköld. *4G: LTE/LTE-Advanced for Mobile Broadband*. Academic Press. Elsevier/Academic Press, 2011.

[9] Trip Denton, Edward Jones, Srini Srinivasan, Ken Owens, and Richard W. Buskens. NAOMI — An Experimental Platform for Multi—modeling. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 143–157, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] E. W. Dijkstra. EWD 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[11] Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit. *Aspect-Oriented Software Development*. Addison-Wesley Professional, first edition, 2004.

[12] Holger Giese, Stefan Neumann, Oliver Niggemann, and Bernhard Schätz. Model-Based Integration. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, chapter 2, pages 17–54. Springer Berlin/Heidelberg, 2011.

[13] George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[14] Rogardt Heldal, Håkan Burden, and Martin Lundqvist. Limits of Model Transformations for Embedded Software. In *35th Annual IEEE Software Engineering Workshop*, Heraklion, Greece, October 2012. IEEE.

[15] Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wąsowski. Guided Development with Multiple Domain-Specific Languages. In Gregor Engels, Bill Opdyke, Douglas Schmidt, and Frank Weil, editors, *MoDELS'07: Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin / Heidelberg, 2007.

[16] Gerd Kainz, Christian Buckl, Stephan Sommer, and Alois Knoll. Model-to-Metamodel Transformation for the Development of Component-Based Systems. In Dorina Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*, pages 391–405. Springer Berlin / Heidelberg, 2010.

[17] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[18] Henrik Lochmann and Anders Hessellund. An Integrated View on Modeling with Multiple Domain-Specific Languages. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 1–10. ACTA Press, February 2009.

[19] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[20] Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *SIMULATION: The Society for Modeling and Simulation International*, 80(9):433–450, September 2004.

[21] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 455–464, Washington, DC, USA, 2003. IEEE Computer Society.

[22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[23] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML^{TM}*. Cambridge University Press, New York, NY, USA, 2004.

[24] Carlos Rodríguez, Mario Sánchez, and Jorge Villalobos. Metamodel Dependencies for Executable Models. In Judith Bishop and Antonio Vallecillo, editors, *TOOLS (49)*, volume 6705 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2011.

[25] Sally Shlaer and Stephen J. Mellor. *Object lifecycles: modeling the world in states*. Yourdon Press, Upper Saddle River, NJ, USA, 1992.

[26] Toni Siljamäki and Staffan Andersson. Performance Benchmarking of real time critical function using BridgePoint xtUML. In *NW-MoDE'08: Nordic Workshop on Model Driven Engineering*, Reykjavik, Iceland, August 2008.

[27] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, chapter 3, pages 57–76. Springer Berlin/Heidelberg, 2011.

[28] Leon Starr. *Executable UML: How to Build Class Models*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[29] Antonio Vallecillo. On the Combination of Domain Specific Modeling Languages. In Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier, editors, *ECMFA*, volume 6138 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2010.

[30] J.B. Warmer and A.G. Kleppe. Building a Flexible Software Factory Using Partial Domain Specific Models. In *Sixth OOPSLA Workshop on Domain-Specific Modeling (DSM'06)*, pages 15–22, Jyvaskyla, October 2006. University of Jyvaskyla.

[31] Thomas Weigert and Frank Weil. Practical Experiences in Using Model-Driven Engineering to Develop Trustworthy Computing Systems. *Sensor Networks, Ubiquitous, and Trustworthy Computing, International Conference on*, 1:208–217, 2006.

Paper 6:
Limits of Model Transformations

Published as:
Limits of Model Transformations for Embedded Software
Rogardt Heldal[1], Håkan Burden[1] and Martin Lundqvist[2]
[1] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
[2] Baseband Research
Ericsson AB
SEW-35
The 35th Annual IEEE Software Engineering Workshop
Iraklio, Greece
13 - 14 October 2012.

1 Introduction

Currently, industrial embedded software applications are often composed of an interacting set of solutions to problems originating from different domains in a relatively complex way. Although these problem domains may be naturally separated, and initially specified by different domain expert designers using different appropriate methods, the actual implementation of the combined application is often delegated to programmers using general programming languages. The programmers are forced to use programming language-dependent details in their manually produced code, including added optimizations for the chosen deployment platform. The result is a mix-up of the desired functionality and structure of the system together with hardware-specific details, all intertwined in the syntax of the programming languages used for implementation.

An answer to this problem is Model-Driven Architecture (MDA; [17, 16]). In MDA the functionality of the system is encoded in platform-independent models that abstract away from hardware- and programming language-specific issues. This way the models become reusable assets across platforms. Besides describing the system, the models can be used to generate a platform-specific implementation through model transformations.

Contribution: We show two challenges for model transformations within the telecom industry; transformations from a functional to an imperative paradigm and reuse of existing transformations for new hardware.

Overview: In section 2 we will further motivate our case study. In section 3 we give the necessary background about the investigated domains and what we consider suitable modeling languages for each domain. In section 4 the delivered application is described together with the implementation process. The outcome of the process is then found in section 5 which is followed by our discussion in section 6. We relate our own findings to previous work in section 7 before we conclude in section 8.

2 Motivation

To what extent are model transformations reusable? According to Mellor *et al.* [16] the vision of MDA is to have reusable transformations for new deployments. On the other hand the MDA Guide [17] delivered by the Object Management Group[24] states that transformations might need adjusting for new deployments. In addition to this conflict among MDA promoters, Mohagheghi and Haugen [18] state that the efficiency and completeness of the generated code are two important aspects when using models in software development. To further investigate the nature of model transformations we evaluate two modeling languages, one functional and one object-oriented, with regards to their code generation capabilities during a project at Ericsson.

The task was to deliver a subsystem of an Ericsson test bed for a 4G telecommunications system, namely the uplink part of an LTE-A radio base station [6]. The requirements on such an application include unconditional real-time performance for calculations on synchronous data and the contextual determination of when signals

[24] http://www.omg.org/

shall be sent and processed as well as operational reliability on new hardware. Since the system was to be demonstrated at the Mobile World Congress in Barcelona, February 2011, there was a firm dead-line. In this context, hand-written C (or sometimes even Assembly) is the current state-of-the-art when it comes to optimally exploiting the hardware's processors and memory. For platform-independent models to provide the software implementations, it is necessary to be able to generate code from them that performance-wise is comparable to hand-written C code [22].

The calculations of the data were modeled using a new functional modeling language with efficient generation of C code, developed in-house. To model the control flow we opted to use an object-oriented modeling language with proven code generation capabilities. Given the requirements on real-time performance and the firm dead-line this raised two questions:

Research Question 1: Which are the key challenges to consider when generating imperative code from a functional modeling language?

Research Question 2: To what extent is it possible to reuse existing model transformations for new hardware?

3 Background

Before we turn our attention towards the case study we will first define the domains and modelling languages that were considered.

3.1 Domain Definition

For us a domain represents one subject matter which is an autonomous world with a set of well-defined concepts and characteristics [21] that cooperate to fulfill a well-defined interface [20]. This definition of a domain is in line with what Giese *et al.* [8] call a horizontal decomposition and ensures the separation of concerns between the domains [7] as well as information hiding [19]. Each domain can be realised as one or more software components as long as these are described by the same platform-independent modeling language [20].

3.2 The Signal Processing Domain

3.2.1 Signal Processing

Within Ericsson the signal processing domain is characterized by a data processing flow, where program state changes and external interactions are kept at a minimum, while more or less fixed and carefully optimized algorithms filter, convert or otherwise calculate incoming data. In telecommunication applications, signal processing plays a crucial role and the necessary algorithms have to be efficient in order to achieve the performance required on speed and quality.

$$DCT-2_n = \left[\cos\frac{k(2l+1)\pi}{2n}\right]_{0\leq k,l<n}$$

```
dct2 :: DVector Float -> DVector Float
dct2 xn = mat ** xn
  where mat =
  indexedMat (length xn) (length xn)
    (\k l -> dct2nkl (length xn) k l)

dct2nkl n k l =
  cos ((k'*(2*l'+1)*3.14)/(2*n'))
    where (n',k',l') = (intToFloat n,
                        intToFloat k,
                        intToFloat l)
```

Figure 29: The Discrete Cosine Transform matrix in mathematical and Feldspar notation.

3.2.2 Implementing Signal Processing

In order to obtain necessary optimized performance on the intended deployment platform, the implementation of the signal processing has to be padded with non-standardized, hardware specific instructions. These instructions are referred to as intrinsic functions, and specify how the specific hardware should be used, in a stricter way than just by compiling standard C or similar. Although the signal processing solution may be fixed and thoroughly verified, every change in deployment platform will bring on the need for new manual intervention, since optimization might be reached in new ways, using other code styles and intrinsic functions.

Feldspar is a domain-specific language currently developed by Chalmers University of Technology and Ericsson for signal processing [2]. The purpose is to limit the gap between the mathematical notation used in the design of signal processing algorithms and their implementation. Feldspar is embedded in Haskell[25] and has no side-effects. A Feldspar program can be evaluated directly via a Haskell interpreter. There is also a code generator that transforms Feldspar programs into C, since run-time performance plays such an important role within signal processing. In Fig. 29 there is an example of a mathematical matrix multiplication used in signal processing together with the equivalent Feldspar definition [2].

3.3 The Control Domain

3.3.1 Controlling the Flow of Execution

We define the term Control Domain as a part of a software application controlling the flow of execution through internal state machinery responding to external communication. The control domain itself does not contain any complicated algorithmic complexity, instead it controls the order in which things are executed; in our case receiving and sending signals, initiating signal processing routines, and collecting their results.

[25] `http://www.haskell.org/`

3.3.2 Executable and Translatable UML

Previous experiences show that an object-oriented modeling language is well suited for describing the control domain; modeling the interaction with surrounding applications in the system and managing the control and exchange of data between the different parts of the signal processing domain [1].

Executable and Translatable UML (xtUML; [23, 15, 20]) evolved from merging the Shlaer-Mellor method [21] with the Unified Modeling Language(UML[26]). xtUML has three kinds of diagrams, together with a textual action language. The diagrams are component diagrams, class diagrams and state machines. There is a clear hierarchical structure between the different diagrams; state machines are only found within classes, and classes are only found within components. Component diagrams have more or less the same syntax as in UML, but both class diagrams and state machines are more restricted in their syntax in comparison to UML. There is an action language, integrated with the graphical elements by a shared meta model [15]. The number of constructions is deliberately kept small so that there is always an appropriate correspondence in the platform-specific model. This also makes it an unsuitable language for complex algorithms since it has a limited set of data structures and only fundamental mathematical notations.

Since xtUML models have unambiguous semantics validation can be performed within the xtUML model by an interpreter. During execution all changes of the association instances, attribute values and class instance are shown [12] as well as the change of state for classes with state machines in the object model.

4 Case Study

4.1 Context

The application chosen for our case study was part of a larger Ericsson test bed project, already involving legacy and new software and hardware, and also new features. The test bed involved 4G telecommunication baseband functionality, based on EricssonâĂŹs existing LTE products, both base station parts and user equipment parts. The test bed included adding features as well as deploying applications on a new hardware platform, resulting in an LTE-A [6] prototype to be presented at the Mobile World Congress in Barcelona, February 2011.

The test bed project lasted part-time for over one year. Already from the beginning it was emphasized that delivering an application that fulfilled the requirements within deadline was more important than using specific methods or languages.

4.2 Domain Identification

The application considered in this project was identified as a self-contained software component, managing a specific part of the data flow in the LTE-A base station, see Fig. 30. The componentâĂŹs external interfaces were well specified, both in terms

[26] http://www.uml.org/

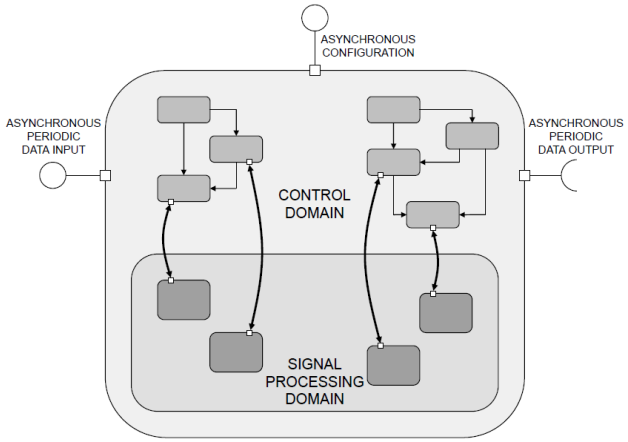

Figure 30: The considered application consisted of the control domain, in turn enclosing the signal processing domain.

of parameter interchange and real-time responsibilities. The application was to be periodically provided with incoming data, while independently requested to update its configuration regarding how to process the incoming data. Upon external triggering in one of these ways, internal chains of data processing was initiated, and expected to run to completion.

4.2.1 Modeling Control using xtUML

Previous experiences with xtUML at Ericsson include reuse of platform-independent models [1] and test generation [4] while still finding the tool easy enough to use by novice modellers [3]. We also knew from experience that xtUML integrates nicely with legacy code written in C.

Since the models are executable it is possible for designers to validate that the models have the required functionality without generating code for deployment. Finally, Ericsson had an existing xtUML-to-C transformation that could compete with hand-written code [22]. We chose BridgePoint[27] as the tool for modeling xtUML.

4.2.2 Modeling Signal Processing using Feldspar

It was decided that it would be a good opportunity to test Feldspar for modeling the signal processing. Feldspar had never been used in an industrial development project before and we wanted to see how well it would fulfill our requirements on real-time performance for the generated code.

[27] `http://www.mentor.com/products/sm/model_development/bridgepoint/`

4.2.3 Modeling the Interface between the Domains

The actual processing of incoming data was dynamic depending on the current configuration. In one instant, a certain configuration would put emphasis on a specific algorithm, quickly changing with a new configuration the next millisecond. Since the number of signal processes and available processors changed from one millisecond to the next it called for the need for independent possibilities for parallelization of each of the involved algorithms, in order to continuously maximize the processing throughput, while keeping the processing latency at a minimum.

The interface between the two domains was defined in an implementation language independent way, identifying parameters necessary for describing the algorithmic and parallelization needs of the different algorithms within the data processing chain. Lochmann and Hessellund refer to such an interface as semantic [14].

4.3 Developers

The people involved in the project had been working between 5 to 15 years each with layer one baseband signal processing in telecommunication equipment at Ericsson. Mainly three developers were involved in the implementation of the models, two implementing the signal processing domain using Feldspar and one developer using xtUML for implementing the control domain. The developers had an unusual combination of expertise in that they were both domain experts and proficient C coders. In addition to the implementors, there was one domain expert in designing signal processing algorithms linked to the project as well as two experts in model transformations; one transformation expert for each modeling language. These two also served as mentors in respective modeling language. Two academic researchers participated as observers and participated in Ericsson AB's internal discussions regarding modelling and model transformations.

4.4 Operation

The operation of the project was inspired by the Scrum method[28], using a backlog and in each sprint there were daily meetings and a burn down chart. At the end of each sprint there was a sprint review and a delivery.

The implementation was done following a component-based development principle [9], where the control domain was viewed as one autonomous component and each algorithm as a component of its own residing inside the control component. Our choice of language for each domain enabled testing of each software component independently and continuously throughout the implementation. When the implementation was complete the models were transformed into target code.

5 Results

Neither the Feldspar nor the xtUML transformation met the requirements of the project and they were both too complicated for those developers without transfor-

[28] `http://www.scrum.org/scrumguides/`

mation expertise to change.

5.1 Transformations across paradigms

Since memory was sparse on the designated platform all variables were limited to 16 bits representation. When transforming Feldspar into C we encountered a problem; the transformation introduced an internal variable for storing an intermediate result. This variable required better precision than the memory constraint allowed, in order to get good enough results 32 bits representation was needed for its encoding. Since this variable was not present in the Feldspar code it could not be marked [16] for exclusive treatment and if all variables were given more memory it would overflow the platform. Due to the time limit the only possibility was to manually change the generated code so that one particular variable accessed an increased memory space.

5.2 Reuse of Transformations

At a crucial moment in the project the xtUML transformation expert was moved to another project taking place in another country. This meant that there was not enough time to change the organization of the project to adapt the existing transformation to the new platform. This adaptation was necessary since the generated C code could not obtain the appropriate utilization of the limited memory and processing capabilities to meet the requirements on real-time performance.

5.3 Transforming the Interface to Multicore

Neither of the existing transformations could adequately handle the multicore parallelization of the platform. This came as no surprise since both lacked a way of specifying the necessary dynamic configurations. The behavior and structure when realizing the interface deployment was therefore coded by hand.

6 Discussion

The possibility for reusing existing solutions is supposed to be one of the strengths of using a model-driven approach to software development [16]. Our case study indicates that this was not the case, at least not when the transformations themselves need to be updated, as was the case in our project. We believe this is particularly true when a new platform is involved. Maybe after a few projects are completed for the same platform, all the relevant platform-specific knowledge is encoded in the transformation. In industrial settings where hardware is continuously changed and updated this is going to be a challenge that MDA has to address for each modeling language that is used within a project.

Hutchinson *et al.* [10] argue that there are too few who have the necessary skills in developing domain-specific languages and efficient transformations within industry. A key issue for successful MDA in industry is the access to transformation experts with short notice since it is not always possible to foresee when a transformation needs to

be updated or optimized. We believe that the importance of the transformations will increase as the number of modeling languages in a project grows. We have probably only started to see the issue of transformation reuse in industry; Lettner *et al.* [13] also report on the problems of porting existing solutions to new platforms in their case study. This is in contrast to the view taken by Kelly and Tolvanen [11] who claim that defining your own modelling language with transformation to code is more efficient than using a general purpose language such as C.

We found that it is vital for an MDE project to have access to the model transformation developers when needed since it is not possible to foresee when a transformation is not going to meet the combined requirements of the project and new hardware. For each modeling language used within a project containing code generation there is going to be one more transformation to adapt and optimize for.

Even if efficient code generation for multicore was not attainable we see some promising possibilities in the inherent properties of the chosen languages. Feldspar functions have no side effects and designed as a library of dynamically composable low-level operations, they would be well suited for execution in a distributed and concurrent manner. Independent instances of state machinery in xtUML can also be deployed as distributed and concurrent threads in a multicore environment.

7 Related work

Hutchinson *et al.* [10] give a general overview of the current industrial practice in model-based software development. As seen in the Discussion, section 6, our experiences relate to their findings on the importance of transformation experts in MDA projects.

Motorola has applied model-driven engineering to describe the asynchronous message passing in a telecommunication system [5]. They also split their system into domains by hierachical decomposition. The difference lies in that while we have used a modeling language to describe the signal processing they have used hand-written code. Another report from Motorola, by Weigert and Weil [24], also found that the transformations had to be adapted to fit new hardware when porting their models. Just as for the previous Motorola report, they do not use models to implement the complex algorithmic behaviour.

8 Conclusion and Future work

The project was a success in terms of meeting deadline and performance, but that was largely due to the fact that the model implementers also had good knowledge of C; the target language of the model transformations. Based on this project we cannot recommend anyone to commence on a similar project without knowing the target language and platform well when considering real time systems running on new platforms. This is due to the fact that the quality of the target code is highly important for performance and it might not be as easy as expected to reuse existing models and transformations.

In our point of view if industry shall succeed in using MDA more research is needed on model transformations, both from an organisational view and the perspective of the efficiency of the generated code. Today, there are usually only a few gurus within companies who have the necessary competence in model transformation [10]. MDA projects will then get too dependent on these transformation experts which creates a bottleneck if they are not continuously accessible during development. Whittle and Hutchinson report on the necessity of educating more software modelers with transformation skills [25]. Based on our own experience we can only agree.

The challenge of generating code across paradigms is another area that needs further exploration. Not only in the case of moving from a functional paradigm to an imperative but also in the case of modeling for multicore and the subsequent transformation to many platforms.

Bibliography

[1] Staffan Andersson and Toni Siljamäki. Proof of Concept - Reuse of PIM, Experience Report. In *SPLST'09 & NW-MODE'09: Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, Tampere, Finland, August 2009.

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169 –178, July 2010.

[3] Håkan Burden, Rogardt Heldal, and Toni Siljamäki. Executable and Translatable UML – How Difficult Can it Be? In *APSEC 2011: 18th Asia-Pacific Software Engineering Conference*, Ho Chi Minh City, Vietnam, December 2011.

[4] Federico Ciccozzi, Antonio Cicchetti, Toni Siljamäki, and Jenis Kavadiya. Automating test cases generation: From xtUML system models to QML test models. In *MOMPES: Model-based Methodologies for Pervasive and Embedded Software*, Antwerpen, Belgium, September 2010.

[5] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Motorola WEAVR: Aspect and Model-Driven Engineering. *Journal of Object Technology*, 6(7):51–88, August 2007. Aspect-Oriented Modeling.

[6] E. Dahlman, S. Parkvall, and J. Sköld. *4G: LTE/LTE-Advanced for Mobile Broadband*. Academic Press. Elsevier/Academic Press, 2011.

[7] E. W. Dijkstra. EWD 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[8] Holger Giese, Stefan Neumann, Oliver Niggemann, and Bernhard Schätz. Model-Based Integration. In Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz, editors, *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, chapter 2, pages 17–54. Springer Berlin/Heidelberg, 2011.

[9] George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[10] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.

[11] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, March 2008.

[12] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[13] Michael Lettner, Michael Tschernuth, and Rene Mayrhofer. A Critical Review of Applied MDA for Embedded Devices: Identification of Problem Classes and Discussing Porting Efforts in Practice. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 228–242. Springer Berlin / Heidelberg, 2011.

[14] Henrik Lochmann and Anders Hessellund. An Integrated View on Modeling with Multiple Domain-Specific Languages. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 1–10. ACTA Press, February 2009.

[15] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[16] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[17] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003.

[18] Parastoo Mohagheghi and Øystein Haugen. Evaluating Domain-Specific Modelling Solutions. In Juan Trujillo, Gillian Dobbie, Hannu Kangassalo, Sven Hartmann, Markus Kirchberg, Matti Rossi, Iris Reinhartz-Berger, Esteban Zimányi, and Flavius Frasincar, editors, *ER Workshops*, volume 6413 of *Lecture Notes in Computer Science*, pages 212–221. Springer, 2010.

[19] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[20] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML™*. Cambridge University Press, New York, NY, USA, 2004.

[21] Sally Shlaer and Stephen J. Mellor. *Object lifecycles: modeling the world in states*. Yourdon Press, Upper Saddle River, NJ, USA, 1992.

[22] Toni Siljamäki and Staffan Andersson. Performance Benchmarking of real time critical function using BridgePoint xtUML. In *NW-MoDE'08: Nordic Workshop on Model Driven Engineering*, Reykjavik, Iceland, August 2008.

[23] Leon Starr. *Executable UML: How to Build Class Models*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[24] Thomas Weigert and Frank Weil. Practical Experiences in Using Model-Driven Engineering to Develop Trustworthy Computing Systems. *Sensor Networks, Ubiquitous, and Trustworthy Computing, International Conference on*, 1:208–217, 2006.

[25] Jon Whittle and John Hutchinson. Mismatches between industry and teaching of model-driven software development. In Marion Brandsteidl and Andreas Winter, editors, *7th Educators' Symposium@MODELS 2011 – Software Modeling in Education*, pages 27–30, Wellington, New Zealand, September 2011.

Appendix D:
Scholarship of Integration

Paper 7:
Are the Tools Really the Problem?

published as:
Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?
Jon Whittle [1], John Hutchinson [1], Mark Rouncefield [1], Håkan Burden [2] and Rogardt Heldal [2]
[1] School of Computing and Communications
Lancaster University
[2] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
MODELS 2013
16th International Conference on Model-Driven Engineering Languages and Systems
Miami, USA
29 September - 4 October 2013.

1 Introduction

When describing barriers to adoption of model-driven engineering (MDE), many authors point to inadequate MDE tools. Den Haan [8] highlights "insufficient tools" as one of the eight reasons why MDE may fail. Kuhn et al. [18] identify five points of friction in MDE that introduce complexity; all relate to MDE tools. Staron [25] found that "technology maturity [may] not provide enough support for cost efficient adoption of MDE." Tomassetti et al.'s survey reveals that 30% of respondents see MDE tools as a barrier to adoption [27].

Clearly, then, MDE tools play a major part in the adoption (or not) of MDE. On the other hand, as shown by Hutchinson et al. [15, 16], barriers are as likely to be social or organizational rather than purely technical or tool-related. The question remains, then, to what extent poor tools hold back adoption of MDE and, in particular, what aspects – both organizational and technical – should be considered in the next generation of MDE tools.

The key contribution of this paper is a taxonomy of factors which capture how MDE tools impact MDE adoption. The focus is on relating tools and their technical features to the broader social and organizational context in which they are used. The taxonomy was developed by analyzing data from two separate studies of industrial MDE use. In the first, we interviewed 19 MDE practitioners from different companies. In the second, we interviewed a further 20 MDE practitioners in two different companies (10 per company). The two studies complement each other: the first is a broad but shallow study of MDE adoption across a wide range of industries; the second is a narrower but deeper study within two specific companies with different experiences of applying MDE. Neither study was limited to tooling issues; rather, they were both designed to capture a broad range of experiences related to MDE use and adoption and, in both, we used qualitative methods to allow key themes to emerge from the data. We focus in this paper only on emergent themes related to MDE tools.

The literature has relatively little to say about non-technical factors of MDE tooling. There have been a number of surveys of MDE tools (e.g., [4, 7, 21]) but they focus on classifying tools based on what technical functionalities they provide. More recently, Paige and Varró report on lessons learned from developing two significant (academic) MDE tools [20]. Again, however, very little is said about understanding users' needs and the users' organizational context: the authors simply state "Try to have real end-users; they keep you honest" and "Rapid response to feedback can help you keep your users."

Indeed, there is a distinct lack of knowledge about how MDE tools are actually adopted in industry and what social and organizational, as well as technical, considerations need to be in place for a tool to succeed. This paper makes a first attempt to redress the balance. Section 2 discusses existing literature on tools, with a focus on understanding users' needs and organizational context. Section 3 describes the methodological details of our studies. Section 4 presents our taxonomy, based on emerging themes from our first study of MDE adoption. Section 5 discusses our second study and relates its findings to the taxonomy. Finally, the paper discusses how the taxonomy can be used to advance research and development of MDE tools (Section 6).

2 Context and Related Work

Tools have long been of interest to those considering the use of technology in industrial settings. In research on computer supported cooperative work (CSCW), there have been two distinctive approaches. On the one hand there are those interested in how individuals use tools and, in particular, how to design tools that are intuitive and seamless to use. This reflects a Heideggerian difference between tools that are 'ready to hand' (they fade into the background) and 'present at hand' (focus is on the tool to the detriment of the 'real' issue) [5] [9, p. 109]. In contrast, another approach, exemplified by Grudin [13] and Brown [3], considers how organizations use tools and argues that failure can be attributed to: a disparity of benefit between tool users and those who are required to do unrecognized additional work to support tools; lack of management understanding; and a failure by designers and managers to recognize their limits. In a comment that might cause some reflection for MDE tool developers, Brown [3] suggests that (groupware) tools are generally useful in supporting existing everyday organizational processes, rather than radical organizational change.

The issue of how software development should be organized and supported has long been discussed and remedies have often, though not always, included particular tools, techniques, and practices. For example, whilst Merisalo-Rantanen et al. [19] found that tools facilitated fast delivery and easy modification of prototypes, amongst the core values of the 'agile manifesto' was a focus on "individuals and interactions over processes and tools" and a number of studies [22] emphasized the importance of organizational rather than technical factors.

However, when considering MDE tools there is little in the way of systematic evaluation. Cabot and Teniente [4] acknowledge MDE tools but suggest that they have several limitations regarding code generation. Selic [23] talks about the important characteristics of tools for the success of MDE, suggesting that some MDE tools "have now reached a degree of maturity where this is practical even in large-scale industrial applications". Recently, Stahl et al. [24] have claimed that MDE does not make sense without tool support. Two studies [1, 18] identify the impact of tools on processes and organizations, and vice versa, but the main focus is on introducing MDE in large-scale software development.

There have been two recent, and very different, studies about the experience of developing and deploying MDE tools. Paige and Varró [20] conclude that: "using MDD tools – in anger, on real projects, with reported real results, is now both feasible and necessary." However, it is significant that this study is about academic MDE tools. In contrast, Clark and Muller [6] use their own commercial experiences to identify lessons learned about tool development, in cases that might be considered technical successes but were ultimately business or organizational failures: "The last decade has seen a number of high profile commercial MDD tools fail ... these tools were expensive to produce and maintain ... there are number of open-source successes but it is not clear that these systems can support a business model". In terms of specific lessons with regard to tools, this one stands out: "ObjeXion and Xactium made comparable mistakes. They were developing elegant tools for researchers, not pragmatic tools for engineers".

3 Study Method

The key contribution of the paper is a taxonomy of MDE tool-related issues. The taxonomy has been developed based on two sets of interviews: a set of 19 interviews from 18 different companies carried out between Nov 2009 and Jul 2010, and a set of 20 interviews carried out in two companies between Jan and Feb 2013. Our method was to use the first set to develop the taxonomy; the second to validate the taxonomy. The two sets are complementary: the first provides broad, shallow coverage of 10 different industrial sectors; the second provides narrow, deep coverage of two companies.

Our first set of interviews is the same set used in earlier publications [15, 16]. However, prior publications gave a holistic view of the findings and did not include data on tools. The procedure for selecting and carrying out the interviews has been described elsewhere [16]. All interviewees came from industry and had significant experience of applying MDE in practice. The interviews were semi-structured, taking around 60 minutes each, and all began with general questions about the participant's background and experience with MDE. All interviews were recorded and transcribed. In total, we collected around 20 hours of conversation, amounting to over 150,000 words of transcribed data.

The second set consists of 10 interviews at Ericsson AB and 10 interviews at Volvo Cars Corporation. The interviewees at Ericsson came from the Radio Base Station unit, which has been involved in MDE since the late 1980s while the interviewees at Volvo represent a new unit that has just started to use MDE for in-house software development for electrical propulsion. The interviews cover more than 20 hours of recorded conversation and were conducted in the same semi-structured fashion as the first set.

Analysis of the interview transcripts was slightly different in each case. The first set was used to develop the taxonomy. Each transcript was coded by two researchers. The initial task was to simply go through the transcripts looking for where the respondents said anything about tools; these fragments were then coded by reference to particular ideas or phrases mentioned in the text – such as 'cost' or 'processes'. The average reference to tool issues per transcript was 11 with 3 being the lowest and 18 being the highest. Inter-coder reliability was computed using Holsti's formula [14], dividing the number of agreements by the number of text fragments. For this research, the average inter-coder agreement was 0.86 (161/187). The researchers then grouped the initial coding into broad themes relating to 'technical', 'organizational' and 'social' issues.

The second set was used to validate the taxonomy. Researchers read the transcripts looking for tool-related issues and then mapped those to the proposed taxonomy. Any deviations from the taxonomy were noted.

4 A Taxonomy of MDE Tool Considerations

This section presents the taxonomy, developed from the first set of interviews. Our analysis process resulted in four broad themes, each broken into categories at two levels of detail: (i) Technical Factors – where interviewees discussed specific technical aspects of MDE tools, such as a missing feature or technical considerations of applying tools in

practice; (ii) Internal Organizational Factors – the relationship between tools and the way a company organizes itself; (iii) External Organizational Factors – influences from outside the company which may affect tool use and application; (iv) Social Factors – issues related to the way people perceive MDE tools or tool stakeholders.

Tables 5-8 form the taxonomy. Each category is briefly defined in the tables, and an example of each sub-category is given. Numbers in brackets are the number of interviewees commented on a particular sub-category (max. 19). Care should be taken when interpreting these numbers – they merely reflect what proportion of our participants happened to talk about a particular issue. They do not necessarily indicate relative importance of sub-categories because one interviewee may have talked in depth about a sub-category whereas another may have mentioned it only briefly. A deeper analysis would be required to produce sub-category weightings. The reader should also avoid the temptation to make comparisons between factors based on the table.

The following subsections present highlights from each theme: we have picked out particularly insightful or relevant experiences from the interview transcripts. We quote from the transcripts frequently; these are given italicized and in quotation marks. Quotes are taken from the transcripts verbatim. Square brackets are used to include contextual information.

The taxonomy is a data-driven, evidence-based description of issues that industrial MDE practitioners have encountered in practice when applying or developing MDE tools. We make no claim that the taxonomy covers all possible tool-related issues; clearly, further evidence from other practitioners may lead to an extension of the taxonomy. We also do not claim that the sub-categories are orthogonal. As will be seen later, some examples of tool use can be classified into multiple sub-categories. Finally, we do not claim that this is the 'perfect' taxonomy. It is simply one way of structuring the emerging themes from our data, and the reader is welcome to re-structure the themes into an alternative taxonomy which better fits his/her purposes.

The taxonomy can be used in a variety of ways. It can be used as a check-list of issues to consider when developing tools. It can be used as a framework to evaluate existing tools. Principally, however, we hope that it simply points to a range of technical, social and organizational factors that may be under-represented in the MDE research community.

4.1 Technical Factors

Table 5 presents the set of categories and sub-categories that relate to technical challenges and opportunities when applying MDE tools. There are six categories.

4.1.1 Category Descriptions

The first, Tool Features, details specific tool functionalities which interviewees felt impacted on project success. These include support for modeling system behavior, architectures, domain-specific modeling, and flexibility in code generation. Code Generation Templates, for example, refers to the ability to define one's own code generation rules, whereas Scoped Code Generation refers to an incremental form of code

Table 5: Technical Categories.

Category	**Sub-Category**
Tool Features *Specific functionalities offered in tools*	- Modeling Behavior (1) - Action Languages (1) - Support for Domain-Specific Languages (6) - Support for Architecture (3) - Code Generation Templates (6) - UML Profiles (1) - Scoped Code Generation (2) - Model Analysis (5) - Reverse Engineering Models (3) - Sketching Models (1) - Refactoring Models (1)
Practical Applicability *Challenges of applying tools in practice*	- Tool Scaleability (1) - Tool Versioning (1) - Chaining Tools Together (2) - Industrial Quality of Generated Code (8) - Flexibility of Tools (3) - Maturity of Tools (1) - Dealing with Legacy (2)
Complexity *Challenges brought on by excessive complexity in tools*	- Tool Complexity (4) - Language Complexity (5) - Accidental Complexity Introduced by Tools (1)
Human Factors *Consideration of tool users*	- Whether Tools Match Human Abstractions (4) - Usability (4)
Theory *Theory underpinning tools*	- Theoretical Foundations of Tools (1) - Formal Semantics (2)
Impact on Development *Impact of tools on technical success criteria*	- Impact on Quality (2) - Impact on Productivity (4) - Impact on Maintainability (3)

generation where only model changes are re-generated. The second category, Practical Applicability, contains issues related to how tools can be made to work in practice. The issues range from tool support for very large models (scaleability), to the impact of using multiple tools or multiple versions of tools together, to the general maturity level of tools and how flexibly they can be adapted into existing tool chains. The third category concerns Complexity, which includes Accidental Complexity, where the tools introduce complexity unnecessarily. The fourth category is Human Factors and includes both classical usability issues but also bigger issues such as whether the way tools are designed (and, in particular, the kinds of abstractions they use) match the way that people think. The final two categories concern the way that the lack of formal foundations leads to sub-optimal tools and the reported perceptions about how tools

impact quality, productivity and maintainability.

4.1.2 Observations

One very clear finding that comes out of our analysis is that MDE can be very effective, but it takes effort to make it work. The majority of our interviewees were very successful with MDE but all of them either built their own modeling tools, made heavy adaptations of off-the-shelf tools, or spent a lot of time finding ways to work around tools. The only accounts of easy-to-use, intuitive tools came from those who had developed tools themselves for bespoke purposes. Indeed, this suggests that current tools are a barrier to success rather than an enabler and "*the fact that people are struggling with the tools... and succeed nonetheless requires a certain level of enthusiasm and competence.*"

Our interviewees emphasized tool immaturity, complexity and lack of usability as major barriers. Usability issues can be blamed, at least in part, on an over-emphasis on graphical interfaces: "*... I did an analysis of one of the IBM tools and I counted 250 menu items.*" More generally, tools are often very powerful, but it is too difficult for users to access that power; or, in some cases, they do not really need that power and require something much simpler: "*I was really impressed with the power of it and on the other hand I saw windows popping up everywhere... at the end I thought I still really have no idea how to use this tool and I have only seen a glimpse of the power that it has.*"

These examples hint at a more fundamental problem, which appears to be true of textual modeling tools as well: a lack of consideration for how people work and think: "*basically it's still the mindset that the human adapts to the computer, not vice-versa*." In addition, current tools have focused on automating solutions once a problem has been solved. In contrast, scant attention has been paid to supporting the problem solving process itself: "*so once the analyst has figured out what maps to what it's relatively easy... However, what the tools don't do is help the analyst figure out what maps to what.*"

Complexity problems are typically associated with off-the-shelf tools. Of particular note is accidental complexity – which can be introduced due to poor consideration of other categories, such as lack of flexibility to adapt the tools to a company's own context. One interviewee described how the company's processes had to be significantly changed to allow them to use the tool: a lack of control over the code generation templates led to the need to modify the generated code directly, which in turn led to a process to control these manual edits. Complexity also arises when fitting an MDE tool into an existing tool chain: "*And the integration with all of the other products that you have in your environment...* " Despite significant investment in providing suites of tools that can work together, this is clearly an area where it is easy to introduce accidental complexity.

It is ironic that MDE was introduced to help deal with the essential complexity of systems, but in many cases, adds accidental complexity. Although this should not be surprising (cf. Brooks [2]), it is interesting to describe this phenomenon in the context of MDE. For the technical categories, in almost every case, interviewees gave examples where the category helped to tackle essential complexity, but also other examples

where the category led to the introduction of accidental complexity. So, interviewees talked about the benefits of code generation, but, at the same time, lamented the fact that "*we have some problems with the complexity of the code generated... we are permanently optimizing this tool.*" Interviewees discussed how domain-specific languages (DSLs) should be targeted at complex parts of the system, such as where multiple disciplines intersect ("*if you have multiple disciplines like mechanical electronics and software, you can really use those techniques*") whilst, at the same time realizing that the use of DSLs introduces new complexities when maintaining a standard DSL across a whole industry: "*their own kind of textual DSL [for pension rules]... And they went to a second company and the second company said no our pension rules are totally different.*" Clearly, as well known from Brooks, there is no silver bullet.

4.2 Internal Organizational Factors

4.2.1 Category Descriptions

Table 6 gives the set of internal organizational categories. The first, Processes, relates to how tools must be adapted to fit into existing processes or how existing processes must be adapted in order to use tools. Tailoring to Existing Processes concerns the former of these; the remaining sub-categories the latter. Sustainability of tools concerns processes for ensuring long term effectiveness of tools, taking into account changes needed to the tools as their use grows within the organization. Appropriation is about how tool use changes over time, often in a way not originally intended. Integration Issues are where new processes are needed to integrate MDE tools with existing tools. Migration Issues are about migrating from one tool to another or from one tool version to another. Offsetting Gains is where a tool brings benefits in one part of the organization but disadvantages in another part of the organization. Maintenance Level is about processes that either mandate model-level changes only, or allow code-level changes under certain constraints. The Organizational Culture category relates to the culture of an institution: to what extent tools need to be adapted to fit culture (Tailoring to Existing Culture), cultural resistance to use new tools (Inertia), a lack of realistic expectations about tool capabilities (Over Ambition), and attitudes that look for quick wins for new tools to prove themselves (Low Hanging Fruit). The third category concerns Skills — both training needs (Training) and how existing skills affect adoption (Availability of Skills).

4.2.2 Observations

Our interviews point to a strong need for tailoring of some sort: either tailor the tool to the process, tailor the process to the tool, or build your own tool that naturally fits your own process. Based on our data, it seems that, on balance, it is currently much easier to do the latter. Some tool vendors actively prohibit tailoring to the process, but rather a process is imposed by the tool for business reasons: "*... the transformation engines are used as services... we don't want to give our customers the source code of the transformation engines and have them change them freely. That's a business question.*"

Table 6: Internal Organizational Categories.

Category	**Sub-Category**
Processes *Adapting tools to processes or vice-versa*	- Tailoring to a Company's Existing Processes (5) - Sustainability of Tools over the Long Term (3) - Appropriating Tools for Purposes They Were Not Designed For (3) - Issues of Integrating Multiple Tools (6) - Migrating to different tool versions (3) - Offsetting Gains: Tools bring gains in one aspect but losses in another (2) - Whether Maintenance is carried out at the Code or Model Level (3)
Organizational Culture *Impact of cultural attitudes on tool application*	- Tailoring to a Company's Culture (4) - Inertia: Reluctance to Try New Things (1) - Over-Ambition: Asking Too Much of Tools (1) - Low Hanging Fruit: Using Tools on Easy Problems First (6)
Skills *Skills needed to apply tools*	- Training Workforce (11) - Availability of MDE Skills in Workforce (4)

When introducing MDE tools, one should think carefully *where* to introduce them. One company reported, "*We needed to find a way to let them incrementally adopt the technology.*" The solution was to first introduce reverse engineering of code into models, as the first part of a process of change management. Another company introduced MDE tools by first using them only in testing. The 'perfect' MDE tool may not always be necessary. For example, one company used MDE where the user interface was not so critical: "*cases which are internal applications . . . where the user interface is not such an issue . . . that's where you get the maximum productivity from a tool like ours.*"

There is a danger, though, in believing that one "killer application" of an MDE tool leads to another: "*prior to that they had used the technology successfully in a different project and it worked and they were very happy, so they thought, ok, this could be applied to virtually any kind of application.*" It is not easy to identify which applications are appropriate for MDE tools and which are not. Apart from obvious industries where MDE has been applied more widely than others (cf. the automotive industry), we do not have a fine-grained way of knowing which MDE tools are appropriate for which jobs.

A curious paradox of MDE is that it was developed as a way to improve portability [17]. However, time and again issues of migration and versioning came up in our interviews: "*[XX] have burned a lot of money to build their own tool which they stopped doing because they lost their models when the [YY] version changed.*"

This migration challenge manifests itself slightly differently as 'sustainability' when considering strategies for long-term tool effectiveness. It was often remarked by our interviewees that an MDE effort started small, and was well supported by tools, but

that processes and tools broke down when trying to roll out MDE across a wider part of the organization: "*the complexity of these little [DSL] languages started to grow and grow and grow... we were trying to share the [code generation] templates across teams and versioning and releasing of these templates was not under any kind of control at all.*" One of our interviewees makes this point more generally: "*One of the things people forget about domain specific languages is that you may be able to develop a language that really is very well suited to you; however, the cost of sustaining just grows and it becomes eventually unacceptable because a language requires maintenance, it requires tooling, it requires education.*"

4.3 External Organizational Factors

4.3.1 Category Descriptions

External organizational factors (Table 7) are those which are outside the direct control of organizations. External Influences include the impact of government or industry-wide standards on the way tools are developed or applied, as well as ways in which marketing strategies of the organization or tool vendors impact on the use and application of tools. Commercial Aspects include how the cost of tools affects tool uptake, how selection of tools can be made based on commercial rather than technical priorities, and how the use of tools relates to a company's business model.

Table 7: External Organizational Categories.

Category	**Sub-Category**
External Influences *Factors which an organization has no direct control over*	- Impact of Marketing Issues (1) - Impact of Government/Industry Standards (4)
Commercial Aspects *Business considerations impacting on tool use and application*	- Business Models for Applying MDE (3) - Cost of Tools (5) - How to Select Tools (2)

4.3.2 Observations

External influences clearly have an impact on whether tools – any kind of tool, not just MDE – are adopted in an organization. Our interviews show that the tool market is focused only on supporting models at an abstraction level very close to code, where the mapping to code is straightforward. This is clearly somewhat removed from the MDE vision. Unfortunately, there is also a clear gap in the way that vendors market their tools and their real capabilities in terms of this low-level approach. As a result, many MDE applications fail due to expectations that have not been managed properly.

Data on the impact of the cost of tools seems to be inconclusive. Some interviewees clearly found cost of tools to be a prohibitive factor. In one case, the high cost of

licenses led a company to hack the tool's license server! For the most part, however, companies do not seem to point to tool costs as a major factor: the cost of tools tends to be dwarfed by more indirect costs of training, process change, and cultural shift: "*. . . it takes a lot of upfront investment for someone to learn how to use the tools and the only reason I learnt how to use them was because I was on a mission.*"

Government or industry standards can both positively and negatively affect whether tools are used or not. MDE tools can help with certification processes: "*they looked at the development method using the modeling tools and said, well, it's a very clear and a very comprehensive way to go and they accepted that.*" In other cases, interviewees reported that MDE tools can make certification more difficult as current government certification processes are not set up to deal with auto-generated code. Sometimes, external legal demands were a main driver for the use of MDE tools in the first place: "*with the European legal demands, it's more and more important to have traceability.*"

4.4 Social Factors

4.4.1 Category Descriptions

When it comes to MDE tools, social factors (Table 8) revolve around issues of trust and control. Tool vendors, for example, have different business models when it comes to controlling or opening up their tools (Interacting with Tool Vendors). Subverting Tools is when a company looks for creative solutions to bring a tool under its control. The data has a lot to say about Vendor Trust, or how perceptions of vendors influence tool uptake. Engineers' Trust also affects tool success: typical examples are when programmers are reluctant to use modeling tools because they do not trust code generated. Career Needs refers to how the culture of the software industry may disadvantage MDE: an example is the ubiquitous use of consultants who are not necessarily inclined to take the kind of long term view that MDE needs.

Table 8: Social Categories.

Category	**Sub-Category**
Control *Impact of tools on whether stakeholders feel in control of their project*	Ways of Interacting with Tool Vendors (2) Subverting Tools: Workarounds Needed to Apply Them (1)
Trust *Impact of trust on tool use and adoption*	Trust of Vendors (4) Engineers' Trust of Tools (6) Impact of Personal Career Needs (1)

4.4.2 Observations

At a very general level, our data points to ways in which different roles in a development project react to MDE tools. One cannot generalize, of course, but roughly speaking, software architects tend to embrace MDE tools because they can encode

their architectural rules and easily mandate that others follow them. Code 'gurus', or those highly expert programmers in a project, tend to avoid MDE tools as they can take away some of their control. Similarly, 'hobbyist programmers', those nine-to-fivers who nevertheless like to go home and read about new programming techniques, also tend to avoid MDE because it risks taking away their creativity. Managers respond very differently to MDE tools depending on their background and the current context. For example, one manager was presented with a good abstract model of the architecture but took this as a sign that the architects were not working hard enough!

One much-trumpeted advantage of MDE is that it allows stakeholders to better appreciate the big picture. Whilst this is undoubtedly true, there are also cases where MDE tools can cloud understanding, especially of junior developers: "*we'd been using C and we were very clear about the memory map and each engineer had a clear view... But in this case, we cannot do something with the generated code so we simply ask the hardware guys to have more hard disc.*"

Similar implications can arise when companies become dependent on vendors. Vendors often spend a lot of time with clients customizing tools to a particular environment. But this can often cause delays and cost overruns and takes control away from the client: "*And suddenly the tool doesn't do something expected and it's a nightmare for them. So they try to contact the vendor but they do not really know what's going on, they are mostly sales guys.*"

MDE asks for a fundamental shift in the way that people approach their work. This may not always be embraced. One example is where MDE tools support engineers in thinking more abstractly, and, in particular, tackling the harder business problems. But engineers may not feel confident enough to do this: "*when you come to work and you say, well, I could work on a technical problem or I could work on this business problem that seems not solvable to me, it's really tempting to go work on the technical stuff.*" MDE tools require up-front investment to succeed and the return on this investment may not come until the tool has been applied to multiple projects. There is a tension here with the consultancy model which is often the norm in MDE: "*So they felt that, let me do my best in this one project. Afterwards, I am moving into some other project... [in a] consultancy organization, you measure yourself and you associate yourself with things in a limited time.*"

5 A Study of MDE Practice in Two Companies

This section presents insights from our second set of data: 20 additional interviews in Ericsson AB and Volvo Cars. Interviewees at Ericsson were users of Rational Software Architect RealTime Edition (RSA/RTE). At Volvo Cars, interviewees used Simulink. This set of interviews was carried out independently of the development of the taxonomy. The taxonomy was used in coding the second set of transcripts but any deviations from the taxonomy were noted.

5.1 Technical Factors

The second study clearly shows that MDE tools can both reduce and increase complexity. Ericsson employees found benefits of using RSA/RTE because of the complex aspects of the radio base station domain, such as synchronous/ asynchronous message passing: *"It takes care of these things for you so you can focus on the behavior you want to have within a base station."* Interestingly, this interviewee has now moved to a new project where all development is done using C++ and a lot of time is spent on issues that were dealt with by the tool before. And it is a constant source of error. On the other hand, *"I don't think you gain advantage in solving all kinds of problems in modeling."* There is a danger of over-engineering the solution: *"You would try to do some smart modeling, or stuff and you would fail. After a while you would end up in a worse place than if you had done this in C++"*.

5.2 External Organizational Factors

Both companies illustrate how external organizational factors impact on MDE success. The functionality of Ericsson's radio base stations is accessed by Telecoms companies such as AT&T through an API. The API is developed using RSA/RTE by 7-8 software engineers. The changes to the API are managed by a forum which is responsible for ensuring that the accepted changes are consistent and that they make sense for the customers: *"We do have a process for how to change it and we review the changes very carefully. For new functions, we want it to look similar, we want to follow certain design rules and have it so it fits in with the rest."* This example illustrates how MDE can be effectively used to manage external influences: in this case, Ericsson models the API as a UML profile and manages it through MDE.

At Volvo, the automotive standard AUTOSAR[29] has made the choice of development tool a non-issue; Simulink is the standard tool: *"... a language which makes it possible to communicate across the disciplinary borders. That the system architect, the engineer and the tester actually understand what they see."*

5.3 Internal Organizational Factors

One Ericsson employee notes the importance of internal organizational support for MDE tools: *"Tool-wise I was better off five years ago than I am today... then we had tool support within the organization. And they knew everything. Today, if I get stuck there is no support to help me."* The quote comes from a system architect at Ericsson who concludes that the tools are difficult to use since they are so unintuitive. The threshold for learning how to produce and consume models can be overcome but it requires an organization where developers are not exposed to different tools between projects.

According to another employee at Ericsson, it is necessary to change the existing processes and culture in order to make the most out of MDE tools: *"I think actually that the technology for doing this [MDE] and the tools, as the enablers, they are more*

[29] AUTomotive Open System ARchitecture; www.autosar.org/

advanced than the organizations that can use them ... Because the organizations are not mature to do it there are few users of those tools and then the usability is poor."

At Volvo a substantial effort has been made in order to enable the transition from Simulink as a specification and prototype tool into a code generation tool; due to the properties of the code generator different design rules are suitable for readability versus code generation. Migrating from one tool to another also requires that old processes are updated: *"When it comes to TargetLink – a competitor to Simulink – we have the knowledge of good and bad design patterns. For Simulink, that is something we are currently obtaining, what to do and not, in Simulink models."*

5.4 Social Factors

It seems that the effort put into tailoring the tools to the existing organization has paid off at Volvo since the domain experts trust the tools to deliver: *"I do like it. In quite a lot of ways. Especially for the kind of software we are developing. It's not like rocket science, really. It's like systems where you have a few signals in, you should make a few decisions, make some kind of output. It is not that difficult applications. There are no complex algorithms... And for that I think Simulink is very sufficient... I like it."*

At Ericsson, interviewees commented that the main difference between working with RSA/RTE and code is that the latter is well-documented on the web: *"You can find examples and case studies and what not in millions."* But when searching for tool-specific help on UML, *"you basically come up empty-handed."*

5.5 Taxonomy Validation

The study at Ericsson and Volvo is in itself revealing about MDE practice. However, for the purposes of this paper, it serves primarily to validate our taxonomy. In only one case did we find that an extension to the taxonomy was necessary. This was on the role that an open community can play in supporting MDE. As discussed in Section 5.4, the lack of online support forums for MDE can lead to feelings of isolation and, in turn, lack of engagement with MDE. We therefore extend our taxonomy to reflect this – by adding a new category, Open Community, with sub-category, Developer Forums, in Table 8. The other issue is that it can be difficult to pick a single sub-category to which a statement applies. Often, a single statement overlaps multiple sub-categories. This, however, was not unexpected. Issues of MDE adoption and tool use are complex and involve many dependencies, so it would be unrealistic to expect a taxonomy with completely orthogonal sub-categories.

6 Discussion and Conclusions

Through two separate studies of MDE practitioners, comprising a total of 39 interviews, we have developed a taxonomy of technical, social and organizational issues related to MDE tool use in practice. This taxonomy serves as a checklist for companies developing and using tools, and also points to a number of open challenges for

those working on MDE tool development. We now discuss some of these challenges, which have emerged from the data.

Match tools to people, not the other way around. Most MDE tools are developed by those with a technical background but without in-depth experience of human-computer interaction or business issues. This can lead to a situation where good tools force people to think in a certain way. We recommend that the MDE community pay more attention to tried-and-tested HCI methods, which can help to produce more useful and usable tools. There is empirical work on studying MDE languages and tools, but this is rarely taken into account.

Research should avoid competing with the market. The research community should focus on issues not already tackled by commercial vendors. Our study found that the majority of tools support the transition from low level design to code. However, many bigger issues of modeling – such as support for early design stages and support for creativity in modeling – are relatively unexplored.

Finding the right problem is crucial. Our studies suggest that finding the right place for applying MDE is a crucial success factor. However, there is very little data about which parts of projects are good for MDE and which are not. Nor is there data about which tools are right for which jobs. In general, even the research community has not clearly articulated how to decide what to model and what not to model, and what tools to use or not to use.

More focus on processes, less on tools. The modeling research community focuses a lot on developing new tools and much less on understanding and improving processes. A particular case is the importance of tailoring. Very little research has been carried out on how best to tailor: what kinds of tailoring go on, how tools can or cannot support this, and how to develop simpler tools that can fit into existing processes with minimal tailoring.

Open MDE Communities. There is a distinct lack of open MDE developer forums. Those who do take the plunge with MDE are left feeling isolated, with nowhere to go to get technical questions answered or to discuss best practice. There are few examples of 'good' models online which people can consult, and efforts towards repositories of such models (cf. [10]) have achieved limited success. There is a chicken-and-egg dilemma here: if MDE is widely adopted, developer communities will self-organize; if it is not, they will not.

The big conclusion of our studies is that MDE can work, but it is a struggle. MDE tools do not seem to support those who try. We need simpler tools and more focus on the underlying processes. MDE tools also need to be more resilient: as with any new method, MDE is highly dependent on a range of technical, social and organizational factors. Rather than assuming a perfect configuration of such factors, MDE methods and tools should be resilient to imperfections.

For the most part, our sub-categories are already known and have been noted either in the literature or anecdotally. France and Rumpe [12], for example, point out that "Current work on MDE technologies tends to focus on producing implementation... from detailed design models". Aranda et al. [1] found that tailoring of processes is critical for MDE. Similarly, Staron found that organizational context has a huge impact on the cost effectiveness of MDE [25]. Indeed, many of our observations about organizational aspects of MDE adoption are not necessarily specific to MDE but are

true of technology adoption generally. However, the contribution of the taxonomy is that it brings all of the factors – both technical and non-technical – together in one place to act as a reference point.

This paper began with the question: "Are tools really the problem?" The answer appears to be both yes and no. MDE tools could definitely be better. But good tools alone would not solve the problem. A proper consideration of people and organizations is needed in parallel. As one of our interviewees noted: "*Wait a second, the tools are really interesting, I agree, but to me it's much more about what is the process and the technique and the pattern and the practice.*"

Acknowledgments The authors would like to thank all those who took part in the interviews, including those who facilitated the study at Ericsson and Volvo.

Bibliography

[1] Jorge Aranda, Daniela Damian, and Arber Borici. Transition to model-driven engineering - what is revolutionary, what remains the same? In France et al. [11], pages 692–708.

[2] Frederick P. Brooks Jr. *The mythical man-month – essays on software engineering (2nd ed.)*. Addison-Wesley, 1995.

[3] Barry Brown. The artful use of groupware: An ethnographic study of how Lotus Notes is used in practice. *Behavior and Information Technology*, 19(4):263–273, 1990.

[4] Jordi Cabot and Ernest Teniente. Constraint support in MDA tools: A survey. In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 256–267. Springer, 2006.

[5] Matthew Chalmers. A historical view of context. *Computer Supported Cooperative Work*, 13(3):223–247, 2004.

[6] Tony Clark and Pierre-Alain Muller. Exploiting model driven technology: a tale of two startups. *Software and System Modeling*, 11(4):481–493, 2012.

[7] João de Sousa Saraiva and Alberto Rodrigues da Silva. Evaluation of MDE tools from a metamodeling perspective. In Keng Siau and John Erickson, editors, *Principal Advancements in Database Management Technologies*, pages 105–131. IGI Global, 2010.

[8] Johan Den Haan. 8 reasons why model-driven approaches (will) fail. `http://www.infoq.com/articles/8-reasons-why-MDE-fails`, 2008.

[9] Paul Dourish. *Where the action is: the foundations of embodied interaction*. MIT Press, Cambridge, MA, USA, 2001.

[10] Robert B. France, James M. Bieman, Sai Pradeep Mandalaparty, Betty H. C. Cheng, and Adam C. Jensen. Repository for model driven development (ReMoDD). In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1471–1472. IEEE, 2012.

[11] Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors. *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012*, volume 7590 of *Lecture Notes in Computer Science*. Springer, 2012.

[12] Robert B. France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In Lionel C. Briand and Alexander L. Wolf, editors, *International Conference on Software Engineering, ICSE 2007, Track on the Future of Software Engineering, FOSE 2007*, pages 37–54, Minneapolis, MN, USA, May 2007.

[13] Jonathan Grudin. Why CSCW applications fail: Problems in the design and evaluation of organization of organizational interfaces. In Irene Greif, editor, *CSCW*, pages 65–84. ACM, 1988.

[14] Ole R. Holsti. *Content Analysis for the Social Sciences and Humanities*. Addison-Wesley Publishing Company, Reading, MA, 1969.

[15] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In Taylor et al. [26], pages 633–642.

[16] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In Taylor et al. [26], pages 471–480.

[17] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[18] Adrian Kuhn, Gail C. Murphy, and C. Albert Thompson. An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development. In France et al. [11], pages 352–367.

[19] Hilkka Merisalo-Rantanen, Tuure Tuunanen, and Matti Rossi. Is extreme programming just old wine in new bottles: A comparison of two cases. *J. Database Manag.*, 16(4):41–61, 2005.

[20] Richard F. Paige and Dániel Varró. Lessons learned from building model-driven development tools. *Software and System Modeling*, 11(4):527–539, 2012.

[21] Jorge Luis Pérez-Medina, Sophie Dupuy-Chessa, and Agnès Front. A survey of model driven engineering tools for user interface design. In Marco Winckler, Hilary Johnson, and Philippe A. Palanque, editors, *TAMODIA*, volume 4849 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2007.

[22] Hugh Robinson and Helen Sharp. The social side of technical practices. In Hubert Baumeister, Michele Marchesi, and Mike Holcombe, editors, *XP*, volume 3556 of *Lecture Notes in Computer Science*, pages 100–108. Springer, 2005.

[23] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[24] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development - technology, engineering, management*. Pitman, 2006.

[25] Miroslaw Staron. Adopting model driven software development in industry – a case study at two companies. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MODELS 2006, Genova, Italy, October 1-6, 2006*, volume 4199 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2006.

[26] Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors. *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*. ACM, 2011.

[27] Federico Tomassetti, Marco Torchiano, Alessandro Tiso, Filippo Ricca, and Gianna Reggio. Maturity of software modelling and model driven engineering: A survey in the Italian industry. In Maria Teresa Baldassarre, Marcela Genero, Emilia Mendes, and Mario Piattini, editors, *16th International Conference on Evaluation & Assessment in Software Engineering, EASE 2012, Ciudad Real, Spain, May 14-15, 2012*, pages 91–100. IET - The Institute of Engineering and Technology, 2012.

Paper 8:
Agile Practices in Automotive MDE

Published as:
Extending Agile Practices in Automotive MDE
Ulf Eliasson [1] and Håkan Burden [2]
[1] Electronic Propulsion Systems
Volvo Car Corporation
[2] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
XM 2013
Extreme Modeling Workshop
Miami, USA
29 September 2013.

1 Introduction

The size and functionality of the software in a modern car increases with each new generation [5] and the software can be executing on in the order of 100 Electronic Control Units (ECUs) spread out in the car. This causes software development to take an increasing part of the total R&D budget for new car models [3]. Automotive manufacturers are traditionally mechanical and hardware oriented companies. The software processes often resemble the traditional waterfall since that works for mechanical development - but it might not necessary be the best for developing software. In a world that moves faster and faster it is crucial to push new features out faster than the competitors. One way that automotive manufactures can speed up their software process is to develop more software in-house, making it possible to iterate their software and introduce new features faster than when ordering the same from a supplier. By introducing MDE we have seen that the domain experts are directly involved in the software implementation and how lead times become shorter.

2 Automotive Software Development at VCC

The system development process at Volvo Car Corporation (VCC) is a waterfall process with a number of integration points throughout where artifacts should be delivered. There are three tracks of parallel development - the software components (SWC), the hardware (known as electronic control units or ECU) and the mechanical parts. Parts of the software is developed in-house while the rest of the software, and most hardware and mechanical systems are developed by sub-contractors. Each iteration of the process has a number of integration steps, as shown in Fig. 31. The system design and signal database (SDB) is the first integration step where the software and its interfaces are designed. Model-In-the-Loop (MIL) is where the implementation Simulink-models are integrated together with other models, either single components together with test models for testing, or in complete MIL with models of different components from different teams. After code generation from the implementation models they are integrated in Hardware-In-the-Loop (HIL) executing on hardware but with the environment around it simulated with models executing in real-time. Finally everything is integrated in a prototype car.

Early on, all the requirements and the system design for the next iteration is captured as a model inside a custom-made tool, referred to as SysTool. The model contains, among many things, software components, their responsibilities as requirements, the software components deployment on ECUs, and the communication between them as ports and signals. The signals are basically periodic data that is sent, and they are described down to the byte size of their data types. The signal definition also includes timing requirements, such as frequency and maximum latency. At a certain point in time the model is frozen and no new changes are allowed until the next iteration starts. These freezes usually last for 20 weeks.

The SysTool model is used for two things. Firstly, signals, ports and their connections and timing requirements are used to implement the SDB. The SDB is used by the software on the ECUs and the internal network nodes to schedule all the signals

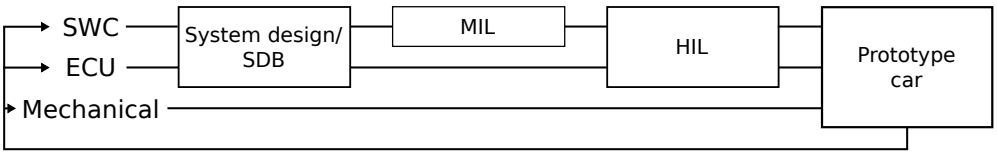

Figure 31: Overall process view.

passing through the networks in the car. The packing and scheduling of the signals is done manually. It takes 9 weeks from a freeze until the SDB is delivered and ready to be used. Secondly, the component model is transformed into Simulink model skeletons. Each Simulink model represents an ECU with skeletons of the deployed software components, including ports and connections. These models are then filled in with functionality by the developers as specified by textual requirements. If the system model changes the Simulink models are updated to reflect this by a click of a button and any implementation already done is kept intact. The implementation must be updated manually if it is dependent on signals that are removed or changed.

The executable Simulink models are tested together with plant models. The plant models represent the environment surrounding the ECUs, including electrical, physical and mechanical systems. This enables the developer to get instant feedback by running and testing the models s/he developed in isolation on their own PCs. The models are also integrated in a virtual car MIL environment where all models developed in-house are integrated and executed. The time frame for getting feedback after delivering a model to the virtual car is counted in days. This gives the developers a possibility to do requirements and design exploration and validation on component level. However, sub-contractors developing software do not normally provide models of their software meaning that where suppliers are developing functionality there are holes in the MIL environment. These holes are filled in by models describing the approximate behaviour. At VCC the development teams are free to choose how to develop their software as they see fit, as long as they can fit their work into the larger overall process and deliver in time. Therefore development within the Simulink models for one ECU can and is conducted agile. However, as soon as there is a need to extend or modify the SDB the developers need to adapt to the overall waterfall process.

The suppliers deliver their software as binaries. Code is generated from the in-house models, built to binary and then the two are linked together. The finished software is loaded on hardware and is then tested on HIL rigs, see Fig 31. Because the supplier only delivers binaries this is the first time that in-house and supplier developed software can be integrated and validated together. Later, software and hardware is integrated with the mechanical systems in a complete prototype vehicle and tested. This is the first time that the whole system is tested together. Each full iteration in Fig 31 has a deadline referred to as a gate and the time between the gates is referred to as E-series.

The use of MDE in the teams makes it possible to break free from the suppliers making it possible to execute and test the in-house software without having to wait for the hardware and software from the suppliers. This enables the team to be agile and work in short iterations.

3 Method

Given that agile practices have shown to be possible at the level of individual teams, we wanted to answer the question:

RQ: Which are the challenges and possibilities for a more agile software development process on a system level?

The question was answered by two exploratory case studies [11]. The main data was collected through two parallel interview series, conducted by the two authors independently of each other. The first set of interviews was launched internally by Volvo Cars in order to identify factors that could improve the existing process. The second study was initiated as a collaboration between Volvo Cars and academia to evaluate and improve the ongoing transition into Model-Driven Engineering, MDE. Both interview sets included eight interviewees, without an overlap between interviewees. In general the interviewees had a background in electronics, physics, automation or mechanical engineering with a limited training in software development from VCC. All interviews were conducted at VCC in Gothenburg, Sweden, and complemented by active participation by one of the authors and on-site observations by the other.

The study focusing on process conducted the first interview in May 2012, the study on MDE started in January 2013. Both sets of interviews were finished by April 2013. Since the interviews were conducted over a longer time frame, collecting and analyzing the data was done in an incremental fashion [11]. From the first interviews of each study a preliminary analysis emerged, identifying themes and concepts that the engineers found challenging in the current combination of process and model-driven implementation. Using a semi-structured format allowed the interviewers to explore new topics as they arose but also to see if spiring hypotheses could be confirmed [9, 12].

The two studies were aligned in May 2013 by a manager who recognized that both lines of inquiry had come to the same conclusion independently of each other, despite the fact that one study was an internal effort to improve the existing process and the other study was an academic collaboration investigating MDE. After the analysis of the interviews had been completed (see section 4) a system architect involved in the scheduling of the SDB was interviewed to reflect on the outcome of the analysis (presented in section 5).

4 Challenges for Agile MDE Practices

From our interviews we discovered a number of issues that have their roots in the waterfall process used on a system level. The most prominent issue, repeated by different interviewees, concerned the SDB in the SysTool. By freezing requirements and system design such a long time before delivery developers are forced to take premature decisions on what data they need to receive or send and where. Because the signals are the interfaces between different components, developed by different teams or sub-contractors, any negative impacts caused by premature decisions are not discovered until late in integration and therefore expensive to change.

The practice of freezing the interface implies that the engineers have to specify the interface they need before they fully understand the internal behaviour of the

component being developed. This means that the interfaces are defined based on the assumptions the developer have at that time and subsequently there are signals that will never be used but still have a share of the limited capacity. This causes two problems. First extra signals need to be scheduled on the network, wasting time for the SDB group in their work as well as causing extra congestion on the network. It also makes it difficult for a developer on an ECU to know which signals are used or not used.

Q: So do you overload the interface? Throw in a signal just in case?

A: Yes, that is what we do. At least I do it [...] and then you end up with the problem knowing which signal it is you should actually use.

As with the spare signals above, developers add extra data-elements to their signals they create to future-proof them. This causes the same problems as with the spare signals but is also harder to redeem because one can't just remove a signal, the signal needs to be modified. It is also harder to check if a data element is used or not compared to seeing if a whole signal is used.

A: Also an old problem we have here at Volvo is that when someone wants to add a new signal, they know it will be hard to change the signal later. So to be prepared they add a few extra data bits to the signal, just in case.

Developers that figure they need to send or receive some data that they did not think about before the freeze do not want to halt their implementation until the next freeze, instead they use existing signals in creative ways. This includes using signals and data-elements in ways that are not described in the requirements making them behave differently than intended. When other groups depend on these signals misinterpretations occur which causes problems.

A: But we have a text document that's about 300 or 400 pages in total if you take all the documents. And that hasn't been updated for a couple of years. So this is wrong. This document is not correct.

Since the textual documentation is inconsistent with the implementation and the interface is overloaded the engineers start to mistrust the artifacts that are supposed to support them in their development. As a consequence one of the other interviewees had developed a work-around for handling that the interface specification was constantly outdated. The solution is to sieve through a second document after the information that concerns the interface being developed and translate that information into a new, temporary, specification.

A: We have in our requirements a list of signals used in the requirement. Now that list is seldom updated. It's hardly ever, so they're always out of date. So I don't actually read them anymore. I just go in through the specific sub-requirements and I read what is asked for my functionality. This is asked. What do I need? I need this and this. So, yeah, so I do it manually, I guess.

The reoccurring theme behind these issues are that developers are forced to make unfounded assumptions about what the SDB will or needs to contain and how the signals in it will be used. Also a shared view between the developers was that it is the development of mechanical and hardware systems and the MDE tools that forced the use of a waterfall process. The issues caused by these assumptions are not found until late in the process with a considerable cost in both time and money as the result.

5 Possibilities for Extending Agile MDE Practices

Based on the results from previous interviews we interviewed an architect responsible for tool and process development at VCC and brought up the identified challenges. He did not see the technical problems of the SysTool and its models to be the main obstacle to overcome, rather it is the culture and the way of thinking in the organization that needs to change.

Q: Why do we have these freezes?

A: We have a traditional gated process, and then you have freezes and gates for everything, period. [...] If we think waterfall it is very logical that we have these freezes, that is what we have to rethink completely. What I'm thinking is that we need to change our system definition process so that it is agile and we can make changes whenever we want [...] and then we can drive this in different speed with different suppliers but it shouldn't be our process that is stopping us.

He also did not think that the hardware development done in parallel requires a waterfall system development process with gates. However, working with suppliers is one reason for having the gated system development process.

Q: How much does it have with working against suppliers and developing hardware?

A: Yes, that is part of the answer. The connection to hardware I do not see as very strong, because the hardware development is not really in the same cycles anyway. Of course, we have hardware upgrades and when it affects the software then it has that connection, if there is some sensor or something that is replaced, but many of these problems are about changes to signals on the network and that is not connected to hardware at all really, at least not at that level. So the hardware is not a big factor. But supplier interaction is of course a factor, because we have a way of working with the supplier where we tell them that on this day we will send the specification and this day you will deliver, and there is a number of weeks between when we send the requirements and they should deliver and then we need a freeze so that we have something to send.

The time between freezing the database until there is a finished implementation of that version, is 9 weeks. Because the developers know that this is their last chance for a while to change the signals they wait until the last minute to put any requests for new or changing signals as close to the freeze as possible. Obviously this results in a lot of change requests to be processed at the same time. It is not until after the freeze that signals and other changes are checked for compatibility and consistency, which might result in a lot of work to make the system design model consistent again.

A: If we say that at this point in time you should submit all your change requests then you get all of these the Friday before instead of getting them more continuously and then you have these weeks of job ahead of you. [...] We need to start thinking about the SDB and frame packing as a product, like any node. So the nodes deliver their software and the SDB delivers the frame packing as a component that integrates with the other stuff. So instead of having a process where the frame packing and everything needs to be finished and done before you can start making a node it should be something that you integrate with the other stuff. [...] Integration is something that already today is happening continuously. It is not a specific day we integrate it is something you try during the E-series and in the end you make it work and then it is time for the next one. [...] At the gate between E-series we do a refactoring of

the SDB, clean it up and pack it. What I think is that, we should continuously allow ourselves to add signals, and also allow each other to add redundant signals. If one signal is wrong we do not remove that signal, because the components are expecting to get this signal. But we can all the time allow ourselves to add signals and therefore we can get double, triple or quadruple signals when we find our way forward. This is roughly as you work with product lines, you allow yourself to over-specify interfaces and so on.

In this way the SDB is tidied up at each gate instead of defined at the system design phase within each iteration.

A: Working like this we can basically end up at releasing a SDB every day throughout the series and as long as a test or E-series environment is alive we can release a SDB daily where we can add new signals to test. Then we can allow ourself to deliver in what frequency we want.

Q: It sounds like most things that need to change are soft issues, are there no technical obstacles for this?

A: No, not really. There is a need for more support in SysTool to make sure that additions that you make are backward compatible. Today this is mostly a manual process. So you need to build in locks so, for this E-series you can only introduce backward compatible changes so that we all the time can export a correct export. And a lot of the stuff that we manually need to clean up is checked automatically. So there are some small tool changes. But this is not the big thing, it is how we think and how we work.

The system model doesn't have to be executable, a non-executable model is enough for the checks that are needed. Also, because the system model is used to generate shells that are filled with executable Simulink models, developers might not see a need or purpose with having more executable models.

A: We will still have these E-series where there will be some refactoring, and to release such an E-series will still take two to three weeks to pack, so it might not be nine weeks but it might be four to five weeks before an E-series release that you need to say what you want in that release. However, the difference from now is that this is not your last chance to get things in it, it is just what will be in it from day one in this series. Then after this you will be able to get things into it as long as it is living.

Because the software development cycles are not bound to the hardware or mechanical development there is no need to follow a similar waterfall process. Also, a more agile system development process would not force the suppliers to be more agile, instead it would make it possible for the teams and their suppliers to work out the best way of working between them. Therefore we can use a more agile process, as already practiced by some of the ECU teams, on a system level. To enable this transition the SysTool needs to support static consistency checks, as proposed by [10], to give the developers and architects confidence in that the changes they make are backward compatible and will not break the integration.

6 Related work

Pernstål [8] has conducted a literature study of lean and agile practices in large-scale software development, such as in the automotive industry, concluding that there is a lack of empirical research within this area.

Eklund and Bosch[6] have developed a method and a set of measures to introduce agile software development in mass-produced embedded systems development, including automotive development. They have discovered that part of introducing agile methods is to gain acceptance in the organization for gradual growth and polishing of requirements instead of using a waterfall approach. They also say that it is possible to have agile software development even though the product as a whole is driven by a plan-driven process.

Kuhn et al. [7] and Arnanda et al. [2] have investigated MDE at General Motors. However, they have not looked at how MDE could change the process and help overcome some of the problems with the traditional process for developing automotive systems.

7 Conclusion and Future Work

The interviewees often thought that the agile challenges were related to MDE or the used tools. However, during the interview with one of the responsible architects for the tools and processes he identified them as caused by the process used on a system level. MDE would be the enabler for a more agile way of working as it provides the teams with a way of testing and iterate their design without having to wait for supplier software or hardware. To get the suppliers on board in such a way of working will take time. But a more agile system development process would not force the teams and suppliers to change their current way of working, but it enables teams and suppliers to agree on a way of working that suits them best instead of forcing them to fit in to the waterfall development process.

We have during our research discovered that agile MDE can be beneficial for automotive development. Using a language close to the domain, such as Simulink, enables engineers trained in the specific domain to work in an environment they recognize and can express their solution in. A model based environment where one can do physical modeling also enables the developers to quickly test their solutions on their own PCs. These are enablers for a more agile development process than is currently the norm in the automotive industry.

We have also discovered that the hardware or mechanical development does not force the software development to follow a waterfall process. The software development is so independent it could have its own process on top of hardware and mechanical development. A more agile software development process would also make it easier to adapt to changes in hardware or mechanical systems.

The interaction with sub-contractors is one of the obstacles that needs to be bridged before agile can happen on all levels. However, a more agile software process on the system level would permit the individual teams and sub-contractors to interact in any way they would think is best instead of forcing them to follow and fit it in to

the waterfall process. Some systems are naturally less appropriate to develop in a completely agile way, such as brakes, and others are so stable and well known that there is little benefit or need for agile development. But having the software process on the system level agile doesn't mean that these domains have to be developed agile, the sub-processes can still be as strict as they need.

The natural thing would be to try to spread the agile MIL environment in all directions and tailor the process [13] to utilize the possibilities of the tools. A first step to enable such a transition would be to allow for faster iterations of the SDB and extending the SysTool to do static consistency checks.

For the future, we plan to implement some of the proposed changes in section 5 to the process at parts of VCC and evaluate them, including looking at how external actors can be integrated in a more agile way of working. Changing a large organization will take time [1]. By starting bottom up we hope that the acceptance for change will be easier to achieve in respect to in-house developers[6] but also for building trust with sub-contractors[4].

Bibliography

[1] Ivan Aaen, Anna Börjesson, and Lars Mathiassen. SPI agility: How to navigate improvement projects. *Software Process: Improvement and Practice*, 12(3):267–281, 2007.

[2] Jorge Aranda, Daniela Damian, and Arber Borici. Transition to Model-Driven Engineering - What Is Revolutionary, What Remains the Same? In *MODELS 2012, 15th International Conference on Model Driven Engineering Languages and Systems*, pages 692–708. Springer, October 2012.

[3] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

[4] Martin Christopher. The agile supply chain: Competing in volatile markets. *Industrial Marketing Management*, 29(1):37–44, January 2000.

[5] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future. *IEEE Computer*, 42(4):42–52, April 2009.

[6] Ulrik Eklund and Jan Bosch. Applying agile development in mass-produced embedded systems. In Claes Wohlin, editor, *Agile Processes in Software Engineering and Extreme Programming*, number 111 in Lecture Notes in Business Information Processing, pages 31–46. Springer Berlin Heidelberg, January 2012.

[7] A. Kuhn, G. C. Murphy, and C. A. Thompson. An Exploratory Study of Forces and Frictions affecting Large-Scale Model-Driven Development. *ArXiv e-prints*, July 2012.

[8] Joakim Pernstål. *Towards Managing the Interaction between Manufacturing and Development Organizations in Automotive Software Development.* PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2013.

[9] Colin Robson. *Real World Research: A Resource for Social Scientists and Practitioner-Researchers.* Regional Surveys of the World Series. Blackwell Publishers, 2002.

[10] Bernhard Rumpe. Agile modeling with the UML. In Martin Wirsing, Alexander Knapp, and Simonetta Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future*, number 2941 in Lecture Notes in Computer Science, pages 297–309. Springer Berlin Heidelberg, January 2004.

[11] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[12] Carolyn B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[13] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial Adoption of Model-Driven Engineering – Are the Tools Really the Problem? In Ana Moreira and Bernhard Schaetz, editors, *MODELS 2013, 16th International Conference on Model Driven Engineering Languages and Systems*, Miami, USA, October 2013.

Paper 9:
Validation through Text Generation

Published as:
Enabling Interface Validation through Text Generation
Håkan Burden[1], Rogardt Heldal[1] and Peter Ljunglöf[1]
[1] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
VALID 2013
5th International Conference on Advances in System Testing and Validation Lifecycle
Venice, Italy
27 October - 1 November 2013

1 Introduction

One way of handling complexity in large-scale software development is to decompose the system into autonomous subsystems that can be independently developed and maintained. In order to successfully integrate the implemented subsystems into a complete and well-functioning system it is necessary to define the connecting points, the interfaces, of the subsystems before or during the implementation [4, 12].

However, the validation of the interfaces is not trivial. For code-centric development it requires an understanding of the used programming languages. In a model-based approach to software development the problem is described by Arlow et. al. [2] as consisting of three main challenges; the necessity to understand the modeling tools used during development, the need to understand and interpret the models that describe the subsystems and their interfaces as well as the underlying paradigm of the models. So, independent of how the subsystems and their interfaces are implemented their validation can only be done by those who understand the implementation. This excludes many of those that have a claim in the delivered system, such as managers and user groups but it also affects many of the system developers. In contrast, textual summaries have the benefit that they can be consumed by all with a claim in the developed software [8].

In an on-going study of model-driven software development for embedded systems in large corporations we discovered that the software engineers had problems with accessing the information encoded in the models in general as well as verifying the correctness of the interfaces in particular. To investigate the effort needed to generate textual summaries from the interfaces we developed a prototype solution, reusing existing software models.

Contribution: First we describe the problem of validating interfaces according to practitioners in industry and why NLG is a possible solution. We then show that the generation of textual summarisations of interfaces can be done with a limited additional effort. The natural language generation technique can be used for other interface specifications that belong to the same modelling language.

Overview: The next section presents the theoretical context of our study, while the practical details are given in Section 3. The results are found in Section 4 and we finish off with a discussion and possibilities for future explorations in Section 5.

2 Theoretical Context

We begin by explaining a few theoretical aspects concerning software models and the specific methodology that we used for generating textual summaries. Related contributions in the area of generating textual summaries from software conclude this section.

2.1 Model-Driven Engineering

One of the aims of using software models is to raise the level of abstraction in order to capture what is generic about a solution. Such a generic, or *platform-independent* [19],

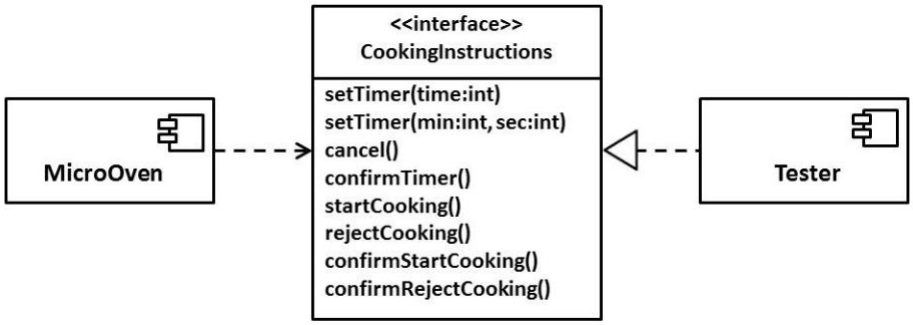

Figure 32: The possible signals between the MicroOven and the Tester listed as an interface.

solution can be reused for describing the same software independent of the platform i.e. the operating system, hardware and programming languages that are chosen to implement the system. The level of detail in the models then depend on if they are to be used as *sketches* giving the general ideas of the system, *blueprints* for manual adaptation into source code or if they are *code generators* and developed with a tool that supports the automatic *model translation* into source code [9]. A condition for the latter is that there is a *metamodel* [19] that specifies the syntactical properties of the modelling language, just as textual programming languages like Java and C have a syntactical specification that can be encoded in BNF [1, 27].

2.2 Executable and Translatable UML

Executable and Translatable UML, xtUML; [16, 20], evolved from merging the Schlaer-Mellor methodology [23] with the Unified Modeling Language, UML. Three kinds of graphical diagrams are used together with a textual *Action language*. The diagrams are *component diagrams*, *class diagrams* and *state machines*. The diagrams are organised hierarchically so that state machines are found inside classes, classes inside components and components can be recursively nested inside other components. The Action language is then used within the diagrams to specify their behaviour and properties. When the models are complete with respect to behaviour and structure they can be automatically transformed into source code through *model translation*.

Components: A component interacts with other components across an interface. An interface declares a contract in form of a set of public features and obligations but not how these are to be implemented. The information and behaviour of the component is only accessible through the specified interface so that the component can be treated as a black box. An example of two components and their interfaces is shown in Figure 32. It consists of two components, MicroOven and Tester where MicroOven provides an interface which Tester depends on.

Class diagrams: For the case of this study it is sufficient to view an xtUML class diagram as equivalent to a UML class diagram.

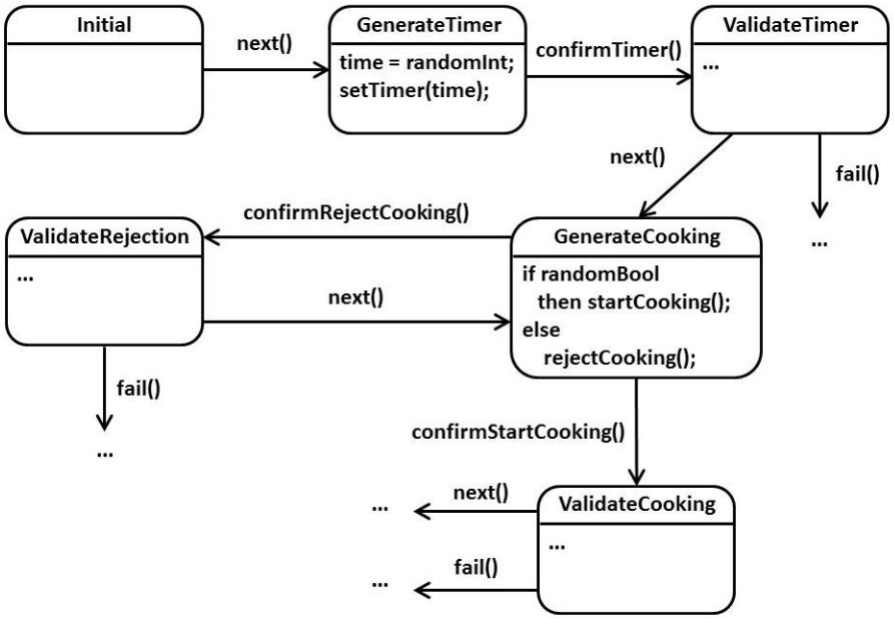

Figure 33: The state machine in Tester describing the validation process.

State machines: In the context of xtUML a state machine is used to model the lifecycle of a class or an object [23]. The transitions between the states can either be defined as internal events or the signals defined by the interface can be mapped onto the transitions so that the external calls change the internal state of the component.

Figure 33 shows the lifecycle of a test object residing inside the Tester component. From the state *Initial* it is possible to reach the *GenerateTimer* state by the internal trigger *next()*. When the external trigger *confirmTimer()* is called the test object is updated and the new state is *ValidateTimer*. Some of the states include pseudo-code to indicate the action to be taken when entering that state.

The semantics of xtUML state machines differ from that of finite-state automata in that the former can interact with their environment as by creating and deleting instances of classes, dispatching events in other state machines and trigger the sending of signals across interfaces etc. Also, if a trigger event does not enable a transition it is not necessarily an error since transition triggers can be ignored if so desired.

Action language: An important property of xtUML is the Action language. It is a textual programming language that is integrated with the graphical models, sharing the same metamodel [23]. Since the Action language shares the same metamodel as the graphical models it can be used to define how values and class instances are manipulated as well as how the classes change their state. Thus we can find Action language within the operations of the classes but it is also used to define the behaviour and the flow of calls through the interface between the components.

Model translation: Code generation is a specific case of model translation with

the aim of translating the model into code. However, model translations can just as well be used for reverse engineering the model into more abstract representations [18]. Model translations are defined according to the metamodel, enabling the same transformations to be reused across domains [19], just like a C compiler is defined on the BNF grammar, not on a specific C program [1]. The models become the code. An example of what the xtUML transformation rules look like and how they can be used is found in Figure 36, which will be further explained in section 3.4.

2.3 Related Work

Previous research has reported on both formal and informal ways of validating the behaviour and structure of software. Examples of formal methods for validating the interfaces are presented by Hatcliff et. al. [11], Mencl [17] as well as Uchitel and Kramer [25] among many. However, they all have similar problems as those mentioned previously by Arlow et. al. [2]; formal methods require a knowledge of the tools, a knowledge of the used models and their paradigm as well as a knowledge of the formal methods.

Lately there has been an increase in the attention towards more informal possibilities for validating software. Spreeuwenberg et. al. [24] argue that if you want to include all stakeholders in the development process you need to have a textual representation of the software models that has the right level of abstraction. In their case they generate a controlled natural language [28] to validate candidate policy decisions for the Dutch Immigration Office.

Another approach towards text generation from platform-independent representations is the translation between the Object Control Language, OCL [26], and English [7, 10]. This work was followed up by a study on natural language generation of platform-independent contracts on system operations [14], where the contracts were defined as OCL constraints that specified the pre- and post-conditions of system operations, i.e. what should be true before and after the operation was executed.

A crosscutting concern is a piece of functionality, such as an algorithm, that is implemented in one or more components. As a result of being scattered across the implementation they are difficult to analyse and when changed or updated it is difficult to estimate how the changes are going to affect the rest of the implementation. Rastkar et. al. [21] argue that having a natural language summary of each concern enables a more systematic approach towards handling the changes. They have therefor implemented a system for generating summaries in English from Java implementations.

There are two previous publications on generating textual descriptions from xtUML. The first describes how natural language specifications can be generated from class diagrams [5] while the second reports on the translation from Action language to English [6] e.g. these publications concentrate on generating textual summaries that describe the internal properties of the components instead of the interaction among components.

3 Case Study

In our collaboration on model-driven engineering with Ericsson AB, Volvo Group Trucks Technology and Volvo Cars Corporation we encountered the problem of validating component interfaces. During our interviews the engineers reported that it was sometimes challenging to validate that the interface was correctly implemented and that the information needed for the validation could be difficult to obtain. As a response we developed a prototype to explore the possibilities to generate natural language summaries for validating component interfaces while keeping the added effort to a minimum.

3.1 Motivation

The interviews were conducted in January, stretching into April 2013. The following interview extract illustrates the problem of understanding the implementation by reading its textual specification.

But we have a text document that's about 300 or 400 pages in total if you take all the documents. And that hasn't been updated for a couple of years. So this is wrong. This document is not correct.

Another issue is that sometimes the engineers are asked to specify the interface before they fully understand the internal behaviour of the component being developed. This means that defining the interface becomes guess work and subsequently there are signals that will never be used but still be given their share of the limited processing capacity.

Q: So do you overload the interface? Throw in a signal just in case?

A: Yes, that is what we do. At least I do it [...] and then you end up with the problem knowing which signal it is you should actually use.

One of the other interviewees had developed a work-around for handling that the interface specification was constantly outdated. The solution is to sieve through a second document after the information that concerns the interface being developed and translate that information into a new, temporary, specification.

We have in our requirements a list of signals used in the requirement. Now that list is seldom updated. It's hardly ever, so they're always out of date. So I don't actually read them anymore. I just go in through the specific sub-requirements and I read what is asked for my functionality. This is asked. What do I need? I need this and this. So, yeah, so I do it manually, I guess.

As a final example of the problems concerning the validation of the component interfaces, a software architect stated that the development tools were difficult to learn and that the development process would be much smoother if there was an accurate textual description of the implementation.

The tools are too unintuitive [...] the threshold for learning how to use them is high [...] but everybody knows how to consume text.

3.2 Aim

Generating text from the implementation should be a suitable solution since it allows the textual description to always be consistent with the implementation as well as understandable by all those with a claim in the project.

The aim of the text generation is a textual description of the intended usage of the interface with as little added effort as possible. For the generation to be feasible in an industrial setting it is beneficial if the generation rules can be maintained and updated without requiring new skills of the engineers. At the same time, the reuse of existing artifacts for generating the summaries will decrease their cost.

Two paragraphs are included in the generated text, one for the intended usage of the interface and one for the unused signals of the interface.

3.3 Setup

The generation is possible due to the reuse of an existing test model, that was adapted from Heldal et. al. [13]. They developed an executable test model for a microwave oven, as illustrated in Figure 32. The test model is designed to capture the intended dialogue between the MicroOven and the user, here represented by the Tester component, as well as its possible error states and constraints. The sequence of the states and transitions therefor follows the process of the MicroOven, with additions for handling erroneous interactions. After the test case is initialised the test-pattern is to generate a signal to the MicroOven across the interface with random values for each parameter. The MicroOven's response is then validated before the Tester transitions into the *next* state in order to generate a new signal with random values. The test case needs to be able to store the results of prior interactions in order to compute the expected value in the validation states and compare that to the given response. If there is a mis-match the test case generates an internal *fail()* event and stores the resulting state so it can be diagnosticised.

After the MicroOven-model has been tested and through the Tester-model they are both translated into platform-dependent source code. The MicroOven is then tested again, now as code by the Tester-code, to see that the intended behaviour of the oven remains the same after deployment. The relationship between the different representations of MicroOven and Tester are depicted in Figure 34.

3.4 Text Generation

Figure 36 shows a fragment of the generation rules. The rules are defined using the Rule Specification Language which are integrated with the xtUML tools [16]. On the first row the signals defined by the `interface` are selected by traversing the concepts of the metamodel according to their relationships. The concept `C_EP` refers to the executable properties of the interface and `C_AS` refers to those executable properties that are signals. The relationships between the concepts are referred to by the unique names `R4003` and `R4004`. Rows 3 and 4 show how the generated text is going to be physically represented [3] as html-pages, using a table since it enables the representation of parallel success paths. All rows that start with a punctuation mark are

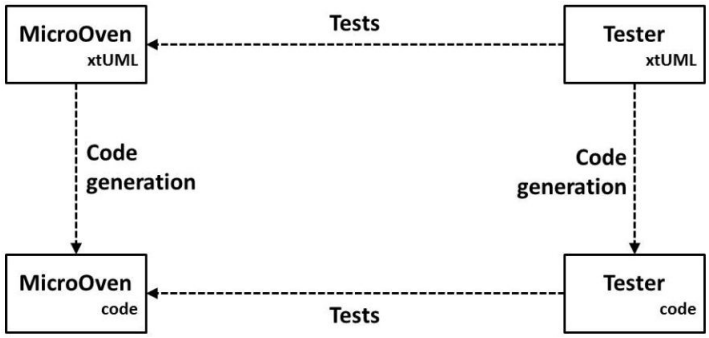

Figure 34: The Tester-model can be reused at code level through code generation.

statements defined by the transformation language while those rows that do not start with punctuation mark define the generated text. The string value of a variable v is obtained by getting its literal text value, `${`v`}`, as in row nine where the signal's name is inserted into the table. Even if the success story only includes those signals that are implemented in the intended usage of the MicroOven the variable `usedSignals` on row five is defined by traversing the entire state machine in order to collect all signals that are used to implement the test case. On row eight the parameters are converted into a textual representation, `paramText`, by calling the function `GetParamData` which is defined by the translation engineer.

In the context of our study it is not relevant to mention in what class the state machine resides that handles the interaction across the interface. That information is excluded in the content selection phase [22] since it is the possible interaction across the interface, as modelled by the state machine, that is interesting, not the internal structure of the components.

For the success path the structure of the text follows the order imposed by the transitions of the state machine, only considering the names of the transitions that constitute the intended usage.

4 Results

The algorithm for navigating through the metamodel to generate the textual summaries is on the size of 100 statements. In comparison a model compiler for generating Java programs consists of 500.000 statements but that covers the entire xtUML definition. Since the state machines and the Action language are so intertwined with the interfaces it is not possible to get a number for the statements needed for translating the interfaces as such into Java. The number of statements for the textual generation is dependent on the present content selection and will increase if more model concepts are to be present in the generation.

In Figure 37 an example of a generated text is shown. It depicts the summarisation of the interface in Figure 32 as implemented by the state machine in Figure 33. The

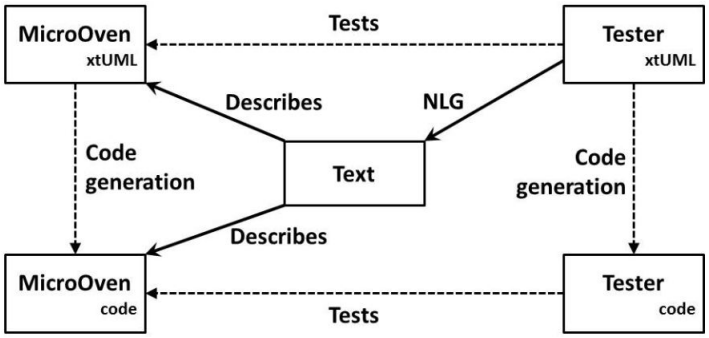

Figure 35: The generated text describes both the xtUML model and the generated code.

```
01 .select many definedSignals related by interface -> C_-
EP[R4003] -> C_AS[R4004]
02 [...]
03 <table border="0">
04 <hr>Unused signals in MicroOven:</hr>
05 .assign unusedSignals = definedSignals - usedSignals
06 .if (not_empty unusedSignals)
07   .for each signal in unusedSignals
08     .invoke paramText = GetParamData
09     <tr><td><i>${signal.name}(paramText)</i></td></tr>
10   .end for
11 .else
12 <b>All defined signals are used.</b>
13 .end if
14 </table>
```

Figure 36: An example of a model-to-text transformation using xtUML.

Intended usage of MicroOven:

1: **Tester** sends *setTimer(time:int)*

MicroOven responds with *confirmTimer()*

2: **Tester** sends *startCooking()*	2: **Tester** sends *rejectCooking()*
MicroOven responds with *confirmStartCooking()*	**MicroOven** responds with *confirmRejectCooking()*

Unused signals in MicroOven:

setTimer(min:int, sec:int)

cancel()

Figure 37: An example of a textual summary.

top-half of the web page shows the intended usage of the oven and the bottom-half details which signals in the interface that are unused in the implementation. The intended usage is given in a table format where alternative usages are given on the same row, side-by-side.

The name of the interacting component is Tester which is carried on to the generated text. This emphasis that naming conventions will affect the readability and understanding of generated texts when they are derived from software implementations. In this case the reading of the generated text would have benefited of naming the testing component to User, who it is meant to represent.

The paragraph for unused signals include *setTimer(min:int, sec:int)* which represents how the interface was overloaded at the point of specification due to the fact that it was unclear how the MicroOven would be used. Since then the decision was taken to specify times in seconds only bu the interface was not changed to reflect this decision. The generated text clearly identifies that there is a mismatch between the specification of the interface and its implementation.

Since the text is automatically generated from the source model it is possible to have a text generated that is consistent with the implementation whenever it is needed; e.g. when considering the implications of adding new functionality, to validate that new functionality conforms to the requirements during implementation or after implementation to understand how the software is intended to be used. This is shown in Figure 35 where the generated text can be used both to describe the original model or the implementation at code-level.

When the model is translated into code the information enclosed in the model is extended with details specific to operative system, chosen programming languages etc. in order to enable the deployment of the generated source code on a specific platform. This added information is then automatically excluded in the generated text since it is not present in the model. The benefit is that the generated text automatically becomes a summary of the interface that focuses on its intended usage while it abstracts away from how the behaviour is obtained. The text can then be reused independent on how the interface is realised on different platforms. As an example, the position of the signal *setTimer(time:int)* in the sequence of intended interactions between the oven

and the user does not depend on if C or Java is used to realise the interface.

The algorithms for generating the success stories and to document unused signals are defined upon the xtUML metamodel. This means that they are reusable across different models that adhere to the metamodel, just as a compiler is defined upon the BNF grammar of programming language and therefor reusable across programs [1].

The structure and naming of concepts and relationships in the metamodel is the main source of complexity in this approach to NLG. Knowing what the concepts and relationships refer to is more challenging than how to map them into a textual representation.

5 Discussion and Future Work

The Object Management Group[30] are the owners of the UML specification and the architects behind the MDA [19, 15] approach to using UML for software development. Their approach for defining the sequencing of the interface signals is to develop a new model, a protocol state machine[31]. Their solution results not only in an additional effort of developing a new model for explaining an old one, but also relies on the same techniques that made it difficult to verify the old model in the first place. As an additional contribution we show how an existing test model can be used for the same purpose as a protocol state machine as well as the source for an NL summary explaining the protocol of the interface.

In relation to previous work on text generation from xtUML our approach does not rely on the understanding of complex linguistic tools. The benefit of only using the same techniques for NLG as for code generation is that there is no additional training cost for companies. This makes it easier to adopt NLG in an industrial context since the number of software engineers with an understanding of both metamodelling and language technology are few. The mapping of metamodel concepts and relationships into linguistic properties will increase the complexity of macro- and microplanning. The drawback of our approach is the limited expressiveness of the transformation rules. For a more varied text structure, less repetitive sentences or for languages with a richer morphology it would be necessary to apply an NLG approach that incorporates linguistic competence. However, striking the balance between richer NLG and what companies are prepared to invest in hiring new competence is yet to be investigated. Our hope is that we will be able to start a new collaboration to enable practitioners in industry to evaluate both the generated texts and the generation procedure. Both lines of query would help to better understand where the balance between cost and readability lies.

Due to the time constraints of the MDE study there was not enough time to gain access to the original models to generate documentation within the industrial context where the issues where found. Instead a prototype was implemented to show how the documentation could be generated from software models with a minimal effort. This opens for another possible route for the future, to further explore the possibilities of NLG for validation purposes in an industrial context. As seen in the related literature

[30] http://www.omg.org/

[31] http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/

there is little involvement from industry to actually use NLG for validation purposes. We believe that this contribution can be a first step to address the problems that engineers actually are facing and as such also open for new ways of adapting NLG and summarisation techniques to the engineer's needs and context.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Pearson Education, Inc., Boston, 2007.

[2] Jim Arlow, Wolfgang Emmerich, and John Quinn. Literate Modelling - Capturing Business Knowledge with the UML. In *Selected papers from the First International Workshop on The Unified Modeling Language UML'98: Beyond the Notation*, pages 189–199, London, UK, 1999. Springer-Verlag.

[3] John Bateman and Michael Zock. Natural Language Generation. In Ruslan Mitkov, editor, *The Oxford Handbook of Computational Linguistics*, Oxford Handbooks in Linguistics, chapter 15. Oxford University Press, 2003.

[4] Antoine Beugnard, Jean-Marc Jézéquel, and Noël Plouzeau. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, 1999.

[5] Håkan Burden and Rogardt Heldal. Natural Language Generation from Class Diagrams. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa 2011, Wellington, New Zealand, October 2011. ACM.

[6] Håkan Burden and Rogardt Heldal. Translating Platform-Independent Code into Natural Language Texts. In *MODELSWARD 2013, 1st International Conference on Model-Driven Engineering and Software Development*, Barcelona, Spain, February 2013.

[7] David A. Burke and Kristofer Johannisson. Translating Formal Software Specifications to Natural Language. In Philippe Blache, Edward P. Stabler, Joan Busquets, and Richard Moot, editors, *5th International Conference on Logical Aspects of Computational Linguistics*, volume 3492 of *Lecture Notes in Computer Science*, pages 51–66, Bordeaux, France, April 2005. Springer Verlag.

[8] Donald Firesmith. Modern Requirements Specification. *Journal of Object Technology*, 2(2):53–64, 2003.

[9] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2004.

[10] Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An Authoring Tool for Informal and Formal Requirements Specifications. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE 2002, Fundamental Approaches to Software Engineering, 5th International Conference*, volume 2306 of *Lecture Notes in Computer Science*, pages 233–248, Grenoble, France, April 2002. Springer.

[11] John Hatcliff, Xianghua Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. Cadena: An integrated development, analysis,

and verification environment for component-based systems. In Lori A. Clarke, Laurie Dillon, and Walter F. Tichy, editors, *Proceedings of the 25th International Conference on Software Engineering*, pages 160–173, Portland, Oregon, USA, May 2003. IEEE Computer Society.

[12] George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[13] Rogardt Heldal, Daniel Arvidsson, and Fredrik Persson. Modeling Executable Test Actors: Exploratory Study Done in Executable and Translatable UML. In Karl R. P. H. Leung and Pornsiri Muenchaisri, editors, *19th Asia-Pacific Software Engineering Conference*, pages 784–789, Hong Kong, China, December 2012. IEEE.

[14] Rogardt Heldal and Kristofer Johannisson. Customer Validation of Formal Contracts. In *OCL for (Meta-)Models in Multiple Application Domains*, pages 13–25, Genova, Italy, 2006.

[15] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture^{TM}: Practice and Promise*. Addison-Wesley Professional, Boston, MA, USA, 2005.

[16] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[17] Vladimir Mencl. Specifying Component Behavior with Port State Machines. *Electronic Notes in Theoretical Computer Science*, 101:129–153, 2004.

[18] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.

[19] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003.

[20] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML^{TM}*. Cambridge University Press, New York, NY, USA, 2004.

[21] Sarah Rastkar, Gail C. Murphy, and Alexander W. J. Bradley. Generating natural language summaries for crosscutting source code concerns. In *27th International Conference on Software Maintenance*, pages 103–112, Williamsburg, VA, USA, September 2011. IEEE.

[22] Ehud Reiter and Robert Dale. Building applied natural language generation systems. *Natural Language Engineering*, 3:57–87, March 1997.

[23] Sally Shlaer and Stephen J. Mellor. *Object lifecycles: modeling the world in states*. Yourdon Press, Upper Saddle River, NJ, USA, 1992.

[24] Silvie Spreeuwenberg, Jeroen Van Grondelle, Ronald Heller, and Gartjan Grijzen. Design of a CNL to Involve Domain Experts in Modelling. In Michael Rosner and Norbert Fuchs, editors, *CNL 2010 Second Workshop on Controlled Natural Languages*, pages 175–193. Springer, 2010.

[25] Sebastián Uchitel and Jeff Kramer. A workbench for synthesising behaviour models from scenarios. In Hausi A. Müller, Mary Jean Harrold, and Wilhelm Schäfer, editors, *Proceedings of the 23rd International Conference on Software Engineering*, pages 188–197, Toronto, Ontario, Canada, May 2001. IEEE Computer Society.

[26] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[27] Manuel Wimmer and Gerhard Kramler. Bridging Grammarware and Modelware. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 159–168. Springer Berlin / Heidelberg, 2006.

[28] Adam Wyner, Krasimir Angelov, Guntis Barzdins, Danica Damljanovic, Brian Davis, Norbert Fuchs, Stefan Hoefler, Ken Jones, Kaarel Kaljurand, Tobias Kuhn, Martin Luts, Jonathan Pool, Mike Rosner, Rolf Schwitter, and John Sowa. On controlled natural languages: Properties and prospects. In Norbert E. Fuchs, editor, *Proceedings of the Workshop on Controlled Natural Language (CNL 2009)*, volume 5972 of *Lecture Notes in Computer Science*, pages 281–289, Berlin / Heidelberg, Germany, 2010. Springer Verlag.

Paper 10:
Model-Driven Engineering at Three Companies

Accepted for publication as:
Comparing and Contrasting Model-Driven Engineering at Three Large Companies
Håkan Burden[1], Rogardt Heldal[1] and Jon Whittle[2]
[1] Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
[2] School of Computing and Communications
Lancaster University
ESEM 2014
8th International Symposium on Empirical Software Engineering and Measurement,
Torino, Italy, September 2014.

1 Introduction

Model-driven engineering (MDE) focuses on the use of high-level abstractions, or models, as primary artefacts for understanding, analyzing and developing software [9]. When applied successfully, MDE can lead to an increase in productivity, better quality software, and more effective reuse of software components [9]. Equally, however, there are many factors – technical, social and organizational – that can cause MDE to hinder a software development effort [19]. In prior work, Hutchinson et al. [6, 7, 8, 19] carried out an extensive study of MDE practice and provided detailed insights as to why some companies adopt MDE successfully whereas others fail. Hutchinson et al.'s study aimed at providing broad coverage of MDE practice: it surveyed 450 MDE practitioners and interviewed 22 MDE professionals from 17 different companies across 9 industrial sectors. The results are revealing; however, such a broad study is also necessarily shallow. The data is based on a survey of mainly closed questions and on semi-structured interviews, where only one or two people per company were interviewed.

To complement the analysis by Hutchinson et al., this paper reports on a study of MDE practice in three large Swedish companies: Ericsson AB, Volvo Cars Corporation and the Volvo Group. The aim was to limit the investigation to a small number of companies but delve more deeply into how MDE is being applied in each. This paper compares and contrasts MDE practice at the three companies and examines how their experiences either validate or refute findings from the earlier study by Hutchinson et al. The three companies in question were deliberately chosen to have both similar and contrasting features. Ericsson's Radio Base Station unit was an early adopter of MDE (since the late 1980s), has many years of experience in applying MDE for mobile communications, and has focused on the use of UML and UML profiles. Electronic Propulsion Systems at Volvo Cars, on the other hand, committed to MDE in 2010 within a domain (automotive) which is often highlighted as a success story for MDE, and focuses on models developed using Simulink. Whereas MDE is now an accepted part of the development culture at Volvo Cars, software implementation at the Volvo Group is still mainly a code-centric business with a few pockets of MDE expertise. All three companies develop products in a specialized embedded systems domain. At the time of the study, all three organizations were undergoing a transition to agile development practices.

Our methodology was to study each company through a series of 25 in-depth interviews with software developers and project managers, split across the companies. Interviews were audio recorded and transcribed. Additional fieldwork, in the form of attending project meetings, informal follow-on conversations, and reporting back to the companies at formal meetings, also provided data in the form of field notes. The data was analysed both inductively for new insights and deductively in comparison to earlier findings.

Our results both confirm and refute findings from Hutchinson et al.'s study and the broader literature on MDE practice. We observed a number of key differences in the way that MDE is being applied across the three companies, and that these differences can have a significant effect on the overall success of the MDE effort. Contrary to perceived wisdom, we have also observed that agile methods and MDE can co-exist

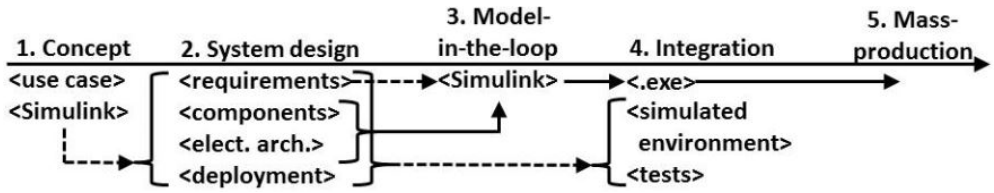

Figure 38: Software development process at Volvo Cars. Solid arrows are automated transformations; dashed arrows are manual implementations.

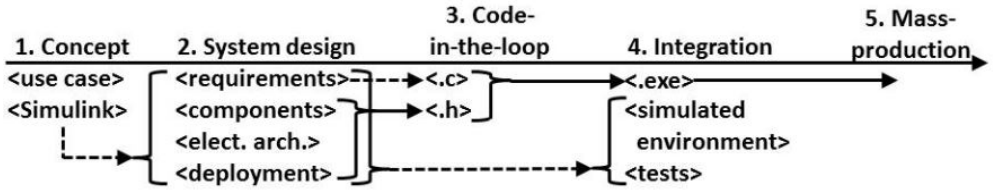

Figure 39: Software development process at Volvo Trucks. Solid arrows are automated transformations; dashed arrows are manual implementations.

peacefully although we note a number of tension points where their joint application may aggravate a development process. The value of abstractions is heavily dependent on context, and can in fact complicate and obscure the structure and behaviour of a system instead of promoting clarity. Where and how to apply MDE still seems to be a question without a fine-grained way to determine the answer. However, in one company, MDE had a significant effect on productivity by integrating domain experts with low-level implementation by forming a new unit supported by an MDE framework.

In section 2 the context of the study is given in more detail while section 3 explains the study method. The analysis is then presented in section 4 and contrasted to earlier work in section 5. The validity concerns are stated in section 6 and finally the conclusion is found in section 7.

2 Company Contexts

Since each company is a separate case in how MDE can be adapted and the aim of the study is to explore the contextual impact on how MDE was adapted – not how MDE was utilized in a specific project – this section supplies an overview of each company.

2.1 Volvo Cars Corporation

Within Volvo Cars Corporation, our study focused on a unit named Electric Propulsion Systems, EPS. EPS is a new unit developing software for electric and hybrid cars. At the time of the study, EPS was developing their second generation of electric propulsion vehicles. Whereas the first generation was developed independently of the

rest of Volvo Cars, the second generation was being integrated into a new Volvo Cars platform under development. The second generation was the first major project at EPS which applied MDE. Although EPS started from scratch in terms of defining and building their MDE infrastructure, there was a legacy of MDE knowledge within Volvo Cars to build on, since another unit, working on combustion engines, has had experience with MDE since 2005. At the time of the study, there were two major process changes underway at EPS. First, there was a push to increase the proportion of in-house software development. This was done in a bid to keep expertise and domain knowledge within the company. Secondly, EPS was undergoing a transition to an agile development process in order to shorten lead-times. In both cases, MDE was seen as a key enabler. To explain how agile methods and MDE are applied at EPS, we briefly explain the development process at Volvo Cars.

Development at Volvo Cars has traditionally followed a waterfall model – see Figure 38. A new concept – e.g., a new engine type – is first scoped out as a set of use cases and Simulink models. Together, these specify the concept. Phase 2 (System design) breaks the concept specification down into a set of textual requirements, component interfaces described in a graphical modeling notation, a database model describing the electrical architecture of the car, and a deployment model describing how to map components on to ECUs (Electronic Control Units), defined in a company-specific tool. An ECU is an embedded system that controls an electrical subsystem in the car (e.g., brake control, transmission control). It is implemented by one or more software components. A modern car can contain up to 150 ECUs and the life-time expectancy of an electrical architecture is at least 10 years. In Phase 2, components are defined black-box. The electrical architecture connects the ECUs to the software components and is defined on top of the AUTOSAR (AUTomotive Open System ARchitecture[32]) standard, a hardware abstraction layer that facilitates updating hardware and software during the electric architecture's lifetime. The system design supports parallel development of components by freezing their interfaces so that components may be developed either in-house or by sub-contractors independently. Given the importance of the component interfaces to the overall development process formal approval for interface changes is only given twice a year.

It is in Phases 3 and 4 (Model-in-the-loop and Integration) where MDE has brought significant changes. Previously, most component development was done by sub-contractors based on the system design.Now, the move is towards defining the internal behavior of components using Simulink in-house. The interfaces of the Simulink models are generated from the system design as black boxes which are then manually filled with behavior according to the textual requirements. The models are then used to generate executables for testing and integration in Phase 4. Since some of the components interact with components that are developed off-site, their interaction is validated in a simulated environment developed by EPS. To calibrate the behavior of the models, testing on rigs of hardware is required. Without the rigs and simulations, validation would have to wait until both in-house and out-sourced components were available for testing. Ultimately, if a concept is successful, it will go into mass production (Phase 5), although most concepts do not make it that far.

[32] `http://www.autosar.org`

As mentioned above, the organization was going through a transition to an agile development process. In practice, this meant that each team working on a component had a large degree of freedom to choose development methods and styles. Agile methods did not, however, extend across all phases. Component interfaces were still defined in Phase 2 with a lengthy approval process required to change them. As we shall see later, this caused tensions for the engineers, who, on the one hand, were working in a very agile fashion, but, at the same time, were expected to work to previously frozen and difficult-to-change interfaces.

2.2 The Volvo Group

Within the Volvo Group, our study focused on Trucks Technology, which develops software for various truck brands – Volvo, Renault, Mack, UD and Eicher. The five brands are developed on top of a shared platform. The truck platform serves the same purpose and has the same overall structure as the car platform at Volvo Cars but has a longer life expectancy (around 20 years). Trucks Technology take overall responsibility for system design and integration but most of the software is implemented by external suppliers. The process at Trucks Technology (Figure 39) is very similar to that at EPS with two exceptions. Firstly, since each truck brand has unique mechanical and hardware configurations (e.g., number of axles, suspension type) each brand needs different parameters and parameter values in the software. With a shared platform for all brands, the system design needs to keep track of approximately 150.00 configurations points. Secondly, at Trucks Technology, Simulink is only used in Phase 1. Implementation is in C. So, instead of generating Simulink from the system design, header files are generated. The specifications of the header files are then realized using C in Phase 3. Even if the implementation is in C, Trucks Technology have developed a simulated environment since they have the same need for early validation and integration of software and hardware as EPS.

After having just successfully launched their first truck on the new platform, at the time of our study, Volvo Trucks was in a time of organizational reflection. Trucks Technology were considering to expand their MDE activities and introduce more executable models, a move that is justified by a need to validate integration earlier and a wish to raise the level of abstraction in their implementation.

2.3 Ericsson AB

The Radio-Base Stations unit (RBS) has used MDE since the late 1980s. RBS delivers software for mobile communications on the GSM, 3G, 4G and multi-standard networks. 90% of the software developed at RBS is deployed on a single hardware node and so there are strict constraints on memory consumption and processing speed. The reason for adapting MDE at RBS was primarily to handle the rising complexity of the products. RBS has defined their own UML profiles using off-the-shelf modeling tools from a major tool vendor[33].

[33]not named, for reasons of confidentiality

All four generations of mobile communication have undergone the same transition, from inception to becoming established products. RBS uses two types of model: descriptive and prescriptive. In the inception phase, models are descriptive – that is, they are used mainly for communication and documentation. In this phase, however, the focus is on developing new features and so models are kept lightweight to avoid slowing the process down. Component diagrams and collaboration or sequence diagrams are used in this case. During the main development phase, implementation may be code-centric or model-centric. In the latter case, prescriptive models are used – that is, models that precisely describe the system to be built. Code may be generated from prescriptive UML state machines and custom-made DSLs are used to test the model. When it is clear that a product will survive on the market and will need to be maintained for many years, the need for more accurate documentation arises, which necessitates creating new models or improving existing models.

A year before the interviews were conducted at Ericsson the interviewees had been reorganized into cross-functional teams, XFTs [16]. At Ericsson, an XFT consists of one software architect (often the domain expert), around five developers and two testers. By combining different skill sets, the aim is to reduce the number of manual hand-overs in the development process as well as to shorten lead times and have a better integration of hardware and software. The transition to XFTs meant that the development teams now have a short-term commitment towards features instead of long-term responsibility for a certain phase of the development process. XFTs have not changed the need for consistent model-based documentation, neither have they enforced a move towards code-centric development. The choice of implementation method is a question of legacy: if a sub-system started as an MDE project, it will continue to be developed using the same MDE tools (or upgraded versions); if the sub-system started out in code-centric fashion, it will remain so.

3 Method

Our study aims to investigate the following two questions: (i) What are the similarities and differences in the way that the three companies have adopted and applied MDE? (ii) How does the nature of practice at the three companies agree or disagree with the findings from Hutchinson et al.?

Recall that our study – as a deep study at three large companies – is intended to supplement that of Hutchinson et al. in order to unveil new insights regarding how MDE is adapted as well as to validate or refute the contributions of Hutchinson et al. For ease of writing, we henceforth refer to Ericsson RBS, Volvo Cars EPS, and Volvo Group Trucks Technology as Radio, Cars and Trucks, respectively.

The three companies were deliberately chosen because they are similar in some respects but different in others. All three are large companies developing products in an embedded systems domain. However, the way they have adopted MDE (see Section 2) is different in each case. These characteristics allowed us to compare and contrast MDE practices across the three companies. Another reason for choosing these companies is more pragmatic: we had existing relationships with key individuals in all cases. We approached our contacts in each company. In the case of Radio and Trucks,

based on a discussion of the aims of the study, our contacts suggested sub-units to study. In the case of Cars, our contact already worked in an appropriate sub-unit, which was therefore self-selected. At the end of this selection process, each company had assigned a point of contact responsible for coordinating the study within the company. This responsibility included selecting suitable interviewees, supporting site visits, and coordinating formal and informal progress meetings within the company.

Data was collected from each sub-unit using a variety of methods: (i) semi-structured interviews with staff; (ii) conversations with staff in informal settings (e.g., over coffee or lunch); (iii) observations of design and development activities, and team meetings; (iv) formal progress meetings with the company points of contact and additional managers or engineers depending on their availability and interest; (v) presentations of results to broader audiences at technical meetings. We used the interview data as the primary source: findings emerged from the interview data; we then used the other sources to validate or elaborate on these findings. Field notes were taken in (ii)-(v); we followed established practices for documenting informal conversations and observations [4, 11]. Data collection was primarily carried out by the first author; the remaining authors participated in activities related to (ii), (iv) and (v). A non-disclosure agreement was signed with each company to allow for open and honest sharing.

We interviewed a total of 25 individuals across the three companies – 9 at Radio, 12 at Cars and 4 at Trucks. Interviewees were selected to represent a range of architects, developers and testers, all of which had been involved in an MDE effort. Table 9 summarizes the distribution of interviewees across these roles as well as the number of years experience each has using MDE in this role. The experience of software development in general correlates to the MDE experience for all interviewees. The relatively low number of interviewees at Trucks is due to the fact that Trucks is not yet active in using MDE during the development phase; hence, we could not interview MDE developers there. At Radio and Trucks, the interviewees were dominated by more experienced engineers (5+ years of MDE) but each company assigned at least one engineer with limited MDE experience. At Trucks, the experienced engineers had often worked with MDE at other companies before joining their present employer. Due to how MDE was adopted at Cars, the less experienced interviewees outnumbered the more experienced ones seven to four. One interviewee at Cars volunteered to participate out of curiosity.

At Cars and Radio, the interviews took place between January and March, 2013. For Trucks, the timing was slightly different – interviews took place between early March and June. None of the interviewees spoke English as their native language. Even so, the majority of the interviews were conducted in English so that the content could be analyzed by all the researchers. All interviews were conducted by the first author and were carried out on-site.

The interviews lasted an hour each on average and were semi-structured in nature. The interviewer began with general questions - such as the interviewee's current role and background - and then delved into MDE-specific topics, such as their experience of MDE, their motivation for applying MDE, the benefits and challenges of applying MDE in practice etc. Each interview followed a different direction according to the interests of the interviewee and what the interviewer picked up as particularly inter-

Role	MDE Exper.	Organization		
		Radio	Cars	Trucks
Architect	5+	4	2	2
	0-5	0	1	0
Developer	5+	2	2	0
	0-5	2	5	1
Tester	5+	1	0	1
	0-5	0	1	0

Table 9: Roles and MDE experience of the interviewees

esting. In some cases, the interview was followed by a second interview or an informal meeting in order to clarify statements. An initial analysis of each interview was conducted immediately after carrying it out; this enabled emerging themes to be explored in subsequent interviews, thus allowing later interviews to confirm or deny earlier ones [13]. Each interview was recorded and transcribed.

The data was analyzed twice looking for how MDE was adapted in the three different organizations. The first analysis was inductive [15], aiming to generate new hypotheses from the data [17]. Each interview was analyzed and annotated by at least two researchers; the annotations were then grouped into themes. Themes, as well as the comments from where they originated, were discussed extensively amongst the three researchers and a consensus was reached on which findings were the most interesting to document. Each finding was validated or further explored using data collected in (ii)-(v) above. After the first analysis was completed, a second, deductive analysis [15] contrasted the findings of Hutchinson et al.'s study with the data collected at the three companies.

4 Findings

In this section, we present the most prominent findings from the data collected during our study. Each subsection describes a recurring theme where we found similarities and/or differences in the way different companies were applying MDE. We illustrate the themes with exemplars taken from the interview data; where it helps to illuminate key issues, quotes are included in italics. These quotes are taken verbatim from the interviews. In some cases, we mark an interviewee's response in a quote with *"I"* and the researcher's question with *"R"*.

4.1 MDE to Bring Development In-House

In earlier work [19], Hutchinson et al. noted that MDE can allow some companies to bring out-sourced development back in-house. The reason given for this in [19] is that out-sourced tasks are often relatively straightforward development jobs; these 'simpler' jobs are easier for subcontractors to carry out independently, but are also easier to automate using MDE. In our new study, we also found that MDE can bring out-sourced jobs back in-house; however, different reasons for this were discovered.

Indeed, one of the primary motivating factors for introducing MDE at Cars was to bring development tasks in-house. Traditionally, the models developed in Phase 2 of Cars' process (Sec. 38) were sent out to external suppliers for implementation. This is currently still the case at Trucks. However, at Cars, MDE has reduced the dependency on these external suppliers since the out-sourced tasks are now done in-house – using Simulink to automatically generate development activities previously left to the suppliers.

Cars have found two key benefits from this. Firstly, Cars now has more control over the development process: they are no longer dependent on suppliers who may not deliver on time and who may not deliver sufficiently high quality components. Secondly, in-house MDE development supports Cars' move towards agile development. In agile development, external dependencies – such as on suppliers – can be a major point of tension. Using suppliers as part of an agile process threatens to break two key agile principles: (1) it slows things down because the team becomes dependent on delivery schedules of the supplier; and (2) it forces the team to write specifications for the supplier to work to, which can lead to overly heavy documentation. Hence, the most effective agile process is one in which all development is carried out in-house (not necessarily physically co-located). MDE has allowed Cars to develop both the Simulink specification and the generated implementations in-house. This has given them a lot more control over their process leading to productivity gains. It is also a success story in which agile and model-driven methods can not only co-exist but can actually mutually support each other.

Compare, for example, the following quotes. At Trucks, which is still dependent on external suppliers in India, there are issues because the Simulink specification sent to India requires domain knowledge to properly understand it. Contrary to perceived wisdom, a formal model is not necessarily understandable without the domain background because there are all kinds of hidden assumptions in the definition and integration of Simulink blocks. This can lead to problems where suppliers without domain expertise improvise incorrectly around the specification: *"... you can just get a feeling for if it works correctly or not. But usually in Bangalore, they don't ... Many of them haven't seen a truck or played around with it at least, so they don't know. They don't have this natural feeling for ... how it should work".*

In contrast, at Cars, bringing the development in-house, using MDE as an enabler, was a way to increase productivity by controlling tightly scheduled iterative development loops: *"That's the key objective, so to say, with this whole in-house software development ... That's where the big increase in speed is from a Volvo perspective."* MDE was also seen to have other benefits than simply speed: it led to cost savings and a higher quality product because Cars could now produce software more relevant to its business, rather than software adapted by a supplier from another customer *"in some cases ... [suppliers] are just reusing an old software that they have delivered to some other OEM [Original Equipment Manufacturer]."*

4.2 Leveraging Domain Experts

A key benefit of MDE at Cars is that it allows domain experts, who specify requirements, to be directly involved in the development process because they understand

the Simulink model and can work together with developers to generate code from it. This again has two major benefits. Firstly, it means that domain experts are much more closely involved in the implementation process: developers and domain experts work together on developing the Simulink models and generating code from them. This in turn leads to a higher quality product. One interviewee put it well: *"In my opinion, the only reason to work with Simulink, is that the system designer and tester and all other stakeholders at Volvo actually understand what they see. You can sit around a Simulink model and discuss an implementation and review it, and so on. If we would have been working in C or something like that, it's just the programmer who understands what is happening in the code."*

Hutchinson et al. noted the importance of having two key skill-sets to make MDE work: both modeling/abstraction skills as well as compiler/optimization developer skills [19]. Both skills are needed to apply MDE successfully in practice because a company has to come up with good models but also typically needs either to develop its own model transformations or adapt off-the-shelf transformations. Whereas Hutchinson et al. recommend both skill sets to be possessed by the individual – what they term the 'MDE guru' – Cars has been successful even when these two skill sets are held by different people. The critical factor that makes this work, however, is that MDE brings the modeling (or domain) expert much closer to the software expert. The use of executable models reduces misinterpretations of the specification and allows domain experts and developers to 'try things out' in close cooperation.

The second benefit is once again that it supports the agile processes at Cars because those specifying the requirements and those developing the implementation work closely alongside each other. Indeed, many of the employees at Cars come from an electrical or mechanical engineering background and have been trained in Matlab/Simulink rather than programming languages like C. Hutchinson et al. [6] found a similar story at another large automotive company in which MDE was introduced precisely to address the lack of software engineers compared to electrical engineers: *"You couldn't find a computer scientist if you went on a search party."* MDE allowed this company to build software using domain experts who understood Simulink and a relatively small team of software specialists could build the generators. The case is similar at Cars. MDE reduces the need for *"pure . . . software guys"* because it allows domain experts to get directly involved in (some) implementation tasks.

4.3 Secondary Software Supporting MDE

Previous research has pointed out that the introduction of MDE tools requires significant effort in adapting them to the context of the organization [19, 18]; commercial MDE tools cannot be simply taken 'off-the-shelf' but may require major tailoring to a company's existing processes. In our study, we found further evidence of the need for this, but also found that it is not just single tools that need tailoring, but that an organization may need an entire suite of supporting applications – which we term secondary software – to make MDE work in practice. Crucially, much of this supporting software is needed precisely to allow domain experts to participate in the development process.

As an example, scripts play a major role in tailoring MDE at Cars: *"they have*

actually just made some scripts to make life easier". These scripts have different origins – *"a few scripts are from the supplier and a few scripts are developed by us, yeah, and by Volvo"* – and are used in various ways, from massaging XML files into the correct format to flashing Simulink models onto hardware or for diagnostic purposes (*"When you generate code and deploy it on hardware, you also through the scripts add a lot of debug information"*). The message is clear: there is a lot of "grunt work" needed around the MDE tools to support both integration of MDE tooling into a broader context and to provide support for domain experts being part of the process. This is an interesting observation because one of the commonly perceived benefits of MDE is to automate 'grunt work.' Here, the grunt work is automated, but the need for it arises directly because of the use of MDE. Kuhn et al. [10] found that one of the frictions that engineers at General Motors encountered was the lack of support for developing scripts and small applications. From our study, it seems that this is a question of organizational support and not the accessibility to domain-appropriate technologies. The motivation for these scripts is to automate recurring tasks that are complicated or tedious. (This supports Hutchinson et al., who state that one of the success stories of MDE is when small DSLs are used "bottom-up" to automate small but repetitive tasks [19]). Furthermore, the scripts form an important part of the secondary software that domain experts at Cars depend on to be active contributors in the development process – since the scripts compensate for their lack of programming skills by automating many tasks that might be straightforward for programmers but not for domain experts. These simple tasks can range from extracting a view of the signals from the overall system model, through to "trimming" a Simulink model to ensure performance in the generated code: *"Yeah. Exactly. So you have a model, and then your inputs to that model are calibration variables. So I've been working with them and optimizing the engine . . . so I get the perfect output . . . "*. Taken together, the scripts provide a chain of functionality, which is absolutely essential: *"But the benefits I perceive from modeling your software, they will not be utilized as much as possible if you don't have a good framework to work with."* In this way the MDE framework encodes the software development process, from requirements down to deployment of binaries and ensures that the information stays in-house. Another consequence is a shift of focus when it comes to reuse – the MDE infrastructure enables fast re-implementation of functionality so the reuse and maintenance of existing models is not as important as the reuse and maintenance of the tool chain. A lesson here is that the framework should developed in an agile way to quickly supply new features as the complexity of the system grows.

In contrast to the emphasis at Cars on developing and maintaining secondary software, Radio, as early adopters of MDE in 1980s, arguably underestimated the need for secondary software to tailor tool suites: *"I: Because that is not the core business for Ericsson . . . R: Perhaps that is something that scares the management at Ericsson, it is a huge invest– I: – yeah. It isn't always that easy to define the business case."* In other words, the development of secondary software can seem like a distraction from the organization's core business. It is essential, however, for companies to recognize that secondary software in MDE *is* part of the core business. The exception at Radio is a UML profile used for defining the management interface for the base stations. The profile has its own organizational unit for its development and maintenance, supported

by a set of tools and model transformations enabling the interface specifications to be ported to different textual formats: *"We need to be very careful with what we change because it will have an impact on customer tools ... We do have a process for how to change it and we review the changes very carefully. For new functions we want it to look similar, we want to follow certain design rules and have it so it fits in with the rest."*.

4.4 Legacy

In adopting new processes in a large organization, a critical challenge is how to deal with legacy. We saw interesting contrasts, for example, in how the three companies integrated new development methods, e.g. agile processes and MDE, with legacy processes, such as a waterfall.

A legacy of the waterfall model at Cars, for instance, is that Cars freezes the component interfaces in Phase 2. The motivation is that sub-contractors and suppliers can then independently develop software in Phase 3 using their own processes and technologies as long as they fulfill the interfaces. In principle, this allows a degree of organizational control over the process. This process of freezing interfaces remains at Cars, despite a move towards agile MDE where components are developed by domain experts using Simulink models and code generation. But a tension point is the need to adhere to frozen interface definitions, which are not easy to change. The interfaces and electric architecture of the car are frozen around twice a year. Although interface changes can be requested between freezes, this involves a lengthy negotiation process between all teams that are dependent on the interfaces and so teams are often reluctant to undertake this: *"R: So you can't change it if you come to understand that, wait a minute, I need to have this parameter as well. I: Absolutely. R: Or I need a new signal to answer this signal in case of - I: Exactly."* Instead, developers try to predict what they will need and add extra interface parameter requests; these parameters may ultimately turn out to be redundant. Hence, there is a problem of interface bloat and Cars ends up with agile teams working effectively but constrained by a highly non-agile infrastructure. The same issue of frozen interfaces was described at Trucks.

Radio uses XFTs, which appears to make the process of changing interfaces straightforward – *"as an XFT team, when I do some changes on the interface, I do both sides"* – because the XFT has control over an entire feature, whereas, previously, developers only had ownership of a small part of a feature and could not update other aspects: *"Somebody had to decide and define down to bits and pieces in a document and assign that work item to you ... And then you had to be finished at the same time because you have to deliver at the same time. So you had to synch that."*

The lesson here seems to be that conflicting process cultures can lead developers to subvert processes: the Cars case of agile developers trying to work around non-agile interface freezes is a case in point.

4.5 System Comprehension and Abstraction

Abstraction is usually argued to be a major benefit of MDE. High-level, abstract models should aid system comprehension and allow stakeholders to see 'the big picture'.

Although raising the level of abstraction of system development is undoubtedly a worthy goal, and MDE certainly goes some way towards supporting this, our data contains numerous examples where models and modeling tools can actually work against system comprehension. Once again, there are both similarities and differences in how this manifests itself at each of the three companies.

Interviewees at Radio, for example, reported multiple problems of comprehension, ranging from overly complex model specifications to difficulties in merging and navigating models to the irreversibility of design decisions when modeling. The sheer size of the models used at Ericsson became a barrier to understanding them: *"We had one model that we always updated and tried not to branch it too much because it became impossible to merge all the changes that we had. And we also had like 8000 sequence diagrams that we should maintain."* Eventually, developers drifted back to Microsoft Word because it was simpler to maintain: *"And I think that got people to be a little bit afraid of the amount, all this big amount of sequence diagrams and all this big amount of information that we need. Of course we needed the information, but it became so painful to update it for every feature that we tried to add to the system ... So I think that is the reason why people started to use Word again."* Another Radio interviewee talked about the difficulty of reversing design decisions: *"So someplace there you have to decide is this capsule actually consisting of many capsules or are there actually two capsules because there are probably parallel state machines. But if the system is analyzed wrongly on top and you try to realize it by using software UML, then it ends up with something that's very, very strange. And it costs a lot to handle that ... Refactoring stuff in that situation is really hard."* The reason for this is that current MDE tools (a UML tool from a major tool vendor in Radio's case) force the developer to make decisions but these cannot easily be undone later because there are so many different models that are mutually dependent on each other across multiple levels: it can be hard to predict the effect of a refactoring and to make sure that all views are consistently updated. This is in contrast to refactoring code because developers are not forced to edit according to the abstract syntax tree: they can cut/paste text freely. Even predicting the effects of changes is hard because it can be very difficult to navigate through models due to multiple levels and viewpoints: *"If you look at it, you just see one point. It's like playing football with a cone. You just see this part. And then you double click to open something else. And double click to open something else. And then you look at that. And then you've forgotten this ... Your brain, my brain, at least, can't handle that."* In addition, the non-linear nature of graphical models makes them difficult to read because there is no obvious place to start: *"When you document, it puts it on the table. Start at page one. And it goes to page 200. And then you're finished. But a model isn't like that."* A similar phenomenon was reported by Kuhn et al. [10], who found that a modeler at General Motors created his own linear numbering scheme for Simulink blocks so he could remember which block to start reading from. The result is that developers at Radio often ended up struggling with the models and the modeling tools. As one interviewee put it: *"Because in textual coding the code is the main artefact. But when you go to graphical modeling, the tool becomes the main artefact. Because you can't really export your models to a new tool."* An open research question, therefore, is how to free MDE from the constraints of tools?

Similar problems were experienced at Trucks. Although MDE adoption at Trucks is still in its early days, there is still a lot of modeling of requirements in Simulink; these Simulink models are typically sent to suppliers for implementation. These requirement models can become very large: *"If we take out the generated specification from this tool ... it is 3,000-plus pages."*. The sheer size of this specification means that suppliers cannot or do not look at everything: *"And he [supplier] says, 'I've never read it. Because I wouldn't do anything but just reading it."'* Typically, Trucks extracts what they think is the relevant part of the specification for each supplier, but this can cause problems if information is missing: *"They're looking through a peep-hole really."* An additional comprehension problem reported at Trucks is related to finding the right level of abstraction. A key issue appears to be that different developers model at different abstraction levels, which can cause mismatches or incorrect interpretations: *"We probably have 60 function developers right now actually working. And they each ... have a little bit of different dialect when they are writing. Somebody goes a little bit lower. Some a little higher. Different experience. And probably you have at least 50 of these persons all writing requirements in this ... area. So you have 50 dialects of writing requirements in this spec."*

4.6 Craftsmanship

One of the key messages that arises from our interviews is a lack of knowledge about when and where to apply MDE. MDE is not suitable for all development tasks – and indeed, different types of MDE are more suitable in some cases than others – but there is currently a lack of experience in the MDE industry as to how to apply MDE most effectively where it matters most. This manifests itself in four ways: (i) knowing which parts of a system are most appropriate for MDE and which are not; (ii) knowing how best to apply MDE once a decision is made; (iii) encoding best practices to transfer MDE "craftmanship" to future projects; (iv) avoiding over-generalization by assuming that if MDE worked well for one project, it will work for others.

Cars has partially overcome some of these problems by adapting the experiences of MDE from other units within Volvo, such as Powertrain, which develops software for combustion engines: *"It's like this. Right now we are in a learning process of what these Simulink patterns look like to generate efficient code. Powertrain, which have been code generating for many years, have patterns for TargetLink, who are competitors, and there Volvo has a very good knowledge of what is good and bad design patterns. When it comes to the world of Simulink, it is something that we currently are obtaining, that experience, what we should and should not do in a Simulink model."* The introduction of Simulink at Cars is therefore based on the concrete experiences from previous projects within the company and the skills of individual engineers. However, this does not imply that Simulink is suitable for all domains within Volvo Cars. Our interviews show, for example, that the transition into MDE is more complicated within the active safety domain. In this case, there are challenging runtime constraints which generated code must satisfy, since the underlying algorithms demand more CPU and memory than the algorithms used at Cars. Even within automotive, some subdomains then are more suitable than others because of (e.g.) the requirements place on the generated code. It appears that as an MDE community, we have very little way

of assessing, in a fine-grained way, the requirements for each sub-domain, how those requirements affect the applicability of MDE, and what are the MDE patterns most appropriate for that particular sub-domain.

Contrast this with the analogous situation for programming languages at Radio. There, the organization has spent many years building up a wide-spread knowledge of how best to match programming languages to specific sub-domains: *"We cannot use one language for every part because they're not suitable for that ... It's great to write test kits in Erlang ... When close to the OSS [Operation Support System] and GUI, of course, Java is excellent."* When transitioning into MDE this knowledge has to be rediscovered. An example of this comes from Cars where a consultant with prior experience of both C and Simulink stated that *"I found out making timers, it was tough in the C code ... it maybe took you 20 minutes to set up one single timer. And in the modeling world, it's just a matter of seconds to make a timer ... On the other hand, if you're doing some data, like moving in an array or something, ... I found that quite complicated in the modeling world. That was more practical in C."* According to one project owner at Radio, there are two questions to ask when considering modeling a certain sub-domain: (i) Will the generated code be efficient enough, and (ii) Do the developers have the right competence: *"You have to take that into account when you form the business case for the start of the project. In some areas, the cost for developing that competence is too big ... If MDE is not something that shows black figures in the end, we shouldn't do it."* Clearly, there are other considerations too (for example, interviewees at all three companies said that applications with strict non-functional requirements on memory handling and processing capacity are known to be difficult to implement using MDE); the point is the application of MDE on a large-scale in practice still appears to be in a relatively immature state – one in which elements of MDE "craftmanship" have not yet been formalized and shared.

4.7 Applying MDE in a New Unit

Other authors have written about strategies for introducing MDE into an organization. Hutchinson et al., for instance, talk of the need to place MDE on the critical path in a development effort – however small it may be – to ensure that the MDE team is not assigned the weakest staff [19]. Aranda et al., in a study of MDE at General Motors, report tensions when engineers were asked to redefine their role in an organization due to the introduction of MDE [1]. This came about because GM employed domain experts with a background in physics and mechanical engineering as software developers, which was made possible because they knew Simulink from their University training. Software engineers were employed to develop the secondary software. This division of resources, which was similarly applied at Cars, required engineers to redefine their roles.

Cars, perhaps by chance, came up with an interesting and rather successful strategy for introducing MDE. Because Electric Propulsion Systems was a new business area for Cars, a completely new unit was created for the development work. The creation of a new unit meant that many of the classical issues associated with change management – such as requiring engineers to redefine their roles, or requiring culture changes within a team – naturally did not exist. The new unit was not tied down to legacy, either

in terms of software, organizational culture, existing process, or external relations. In other words, the new unit more or less had a clean slate to develop new ways of working as needed. Crucially, however, such a new unit could not succeed in a vacuum. It was not enough simply to create the new team and expect great things to happen; rather, the team had access to historical expertise in MDE from other units within Volvo. But the new unit allowed the team to pick up best practices as they wished, and to exclude practices that did not resonate with their new way of thinking. An example of this is how the new unit at Cars defined their own agile processes on top of MDE processes from Powertrain. Although the creation of a new unit may not always be feasible, or even desirable, this way of handling the introduction of MDE is consistent with the recommendations of Christensen, who states that when changes become too extensive within an existing organization the solution is to start afresh [3].

5 Related Work

5.1 Hutchinson et al.

The most closely related work to ours is that of Hutchinson et al. [6, 7, 8, 19]. This subsection therefore gives a detailed comparison of the findings from the two studies. The main conclusions of Hutchinson et al. are given in Table 10, grouped according to different aspects of introducing MDE. The final column in Table 10 indicates whether the findings of Hutchinson et al. are validated (V) or refuted (R). Although most of our results validate Hutchinson, we both (i) offer additional evidence to support the finding (a form of replication), and (ii) typically offer deeper or additional insights related to the finding in each case. Earlier sections of the paper have elaborated on validated findings. Therefore, we focus on findings which refute Hutchinson et al. in this section. The refuted findings come exclusively under the categories of Control and Training.

We discovered that software architects at Radio who are used to specifying their domain knowledge in textual documents found MDE tools and languages intimidating. The introduction of MDE enforced on them a level of formality with which they were not comfortable. In contrast to Hutchinson, who argues that middle managers are often a bottleneck when introducing MDE because they are risk-averse, our study shows that this need not be the case – again at Radio, we found that middle managers can be champions of MDE precisely because it can minimize risks related to human error or external organizational dependencies. We also found that code gurus are not necessarily averse to MDE; software engineers employed at Cars, for instance, do not mind building secondary software to support domain experts because they have specifically been employed as consultants to do this.

Hutchinson et al. have criticized UML education in Universities because educators often tend to focus on teaching the syntax of UML rather than problem solving. At Cars, the domain experts have been trained at University in Simulink and, contrary to UML, appeared to be well-equipped to enter the company and transition, using MDE and Simulink, to a more software development-oriented role. Hutchinson et al. have also argued that Universities unnecessarily separate modeling/abstraction skills from

	Findings from Hutchinson et al.	**V**alidated or **R**efuted
Domain	Successful MDE tends to favor DSLs over General-Purpose Languages [19]	V
	When developing DSLs, focus on narrow, well-understood domains [7, 8, 19]	V
	Domain experts already model: they use informal DSLs so can transition easily to MDE [19]	No data
Process	MDE more successful if driven bottom-up, by developers (not from managers) [6, 19]	V
	Put MDE on a critical path to get the best resources [6, 7, 8, 19]	V
	Productivity gains from MDE can be lost elsewhere (e.g., unreadable generated code) [7, 19]	V
	Most MDE project failures are at scale-up [7, 19]	V
	Organizations must constantly evolve MDE infrastructure as understanding grows [6]	V
Motivation	MDE needs a clear business driver [6, 19]	V
	MDE works well for domain-specific products, rather than general software [8]	V
	Target MDE where it can have maximum impact quickly [8, 19]	V
	Organizational buy-in for MDE is important at all levels [6, 8]	V
Control	MDE requires significant customization of tools to an organization's context [19]	V
	MDE can bring outsourced development activities back in-house [19]	V
	Architects like MDE: the generator lets them control architectural rules more easily [19]	R
	Code gurus do not like MDE: they lose control to the generator [19]	R
	Middle managers are risk-averse and reluctant to try new techniques such as MDE [19]	R
Abstraction	If the models are too close in abstraction level to code, there are limited benefits [6]	V
	Simplicity in models can be counter-productive; managers may see it as laziness [19]	No data
	MDE engineers need a mix of abstraction and compiler skills [8, 19]	V
Training	MDE training is too often focused on tool idiosyncrasies, rather than problem solving [6]	R
	Universities do not adequately train engineers in MDE [19]	R
	Businesses hire too much based on knowledge of specific technologies of the day, which may not be relevant to modeling [19]	R

Table 10: Comparing the conclusions of Hutchinson et al. with data from Radio, Cars and Trucks.

compiler/optimization skills in their curricula. This can cause problems, but does not appear to do so at Cars because Simulink allows the domain experts and implementors to work very closely together. Similarly, at Cars, there is a supply of Simulink-trained engineers leaving University, which is exactly what Cars needs to hire. This conflicts with many situations in the Hutchinson study where companies hired engineers skilled in a particular technology, but then required them to work on modeling, for which they were not trained.

5.2 Other Related Work

There are a very large number of papers describing case studies of applying MDE in practice, as well as a smaller number of empirical studies on industrial use of MDE. To limit the scope, we here describe only the most prominent of these papers, with a particular emphasis on those that go beyond the pure technical aspects of MDE in that they also include organizational and social factors, especially as relates to the automotive and telecoms sectors.

In a recent publication, we presented a taxonomy of challenges and potentials when applying MDE tools in industry [18]. The taxonomy was deduced from the interviews conducted by Hutchinson et al. and validated through a limited set of the data obtained in this study. The findings emphasize how current MDE tools are lacking in both usefulness and usability and concludes that more research is needed to understand the interplay between organizations, human factors and the technical features of the tools.

Kuhn et al. [10] and Aranda et al. [1] report on two parallel studies of applying MDE at General Motors. The former focus on individual perceptions of the adoption of MDE at GM; the latter reports on how MDE induced changes at the organizational level. At the individual level, engineers experienced both forces and frictions related to MDE tools and languages – for example, a lack of support for developing secondary software (see Sec. 4.3). At the organizational level, [1] found, for instance, that software developers felt "down-graded" when MDE was introduced, because they were now asked to focus on secondary software whereas domain experts focused on primary functionality.

Baker et al. [2] describes a process of introducing MDE at Motorola. They also report on the lack of organizational maturity in terms of which processes to use in combination with MDE, the difficulties in adapting existing skills to the new challenges of MDE and the unwillingness of the organization to change in order to make the most out of the transition into MDE. A previous study conducted at Radio can be found in [12]. Here the emphasis is on MDE in relation to architectural concerns (deployment, algorithms, performance etc.) but the authors point out some organizational aspects of MDE, such as the possibility that the cost of modeling can outweigh the benefits.

6 Validity

In terms of threats to internal validity, we followed a systematic approach in setting up the study and best practice guidelines in both data collection and analysis [13, 14, 17].

A systematic approach is straightforward to follow in the case of the interviews as strict protocols can be set up. In the case of the more informal aspects of the fieldwork there are inevitable critiques of representativeness and rigor. We were not able to carry out more rigorous, formal experiments within the companies since we wanted to avoid disruptions. In such cases, informal interactions are an established way to gather data [4, 11]. A benefit of informal interaction is that it lets the respondent be in control, which, in turn, enables the discourse to lead to new topics not anticipated by the researcher [5]. The informal settings allowed the researchers to interact with employees that were not assigned as interviewees. This gave an opportunity to validate if data collected through the formal interviews represented a shared understanding or a minority view. Another benefit of the informal interactions was that often engineers from other units would be present, which gave the opportunity to see if findings carried across to other units or not. We have taken great care in our study to validate emerging findings from the interviews. Throughout the study, the companies have been involved in validating our analysis: in seminars with the respective contact persons and through recurring interactions with engineers in both formal and informal settings. All three companies approved this paper. In terms of threats to external validity, all companies are Swedish developing large-scale software for embedded systems and this software is expected to have a long lifetime. Therefore, our findings may not apply to MDE in smaller companies, other countries or for software with a shorter life expectancy. Another limitation is that we studied only one unit at each company; other units might have very different experiences of MDE. We have mitigated this to some extent by interviewing a broad spectra of engineers within each organization.

7 Conclusions

In general our results validate the conclusions of Hutchinson et al. However, in two ares our data refutes their observations: how to introduce MDE so that the engineers feel they do not loose control and in to which extent engineers have the right training for MDE. Additionally, our own data illustrates how it is possible to involve the domain experts directly in market leading software implementation. The resulting productivity gains are possible due to investments in secondary software that compensates for the domain experts lack of programming skills. In this way the secondary software captures both the domain and the best practices of implementing the same. Our findings present initial insights in how MDE and agile practices can naturally coexist in the context of embedded software. A future direction of research is then to further explore how domain experts can be included not only in the specification of new features but become directly involved in their implementation - from innovative idea to integration on hardware. A special case of the above is how agile practices and MDE can combine to enable innovation in industrial sectors where external organizations play an important role in the overall development.

Bibliography

[1] Jorge Aranda, Daniela Damian, and Arber Borici. Transition to Model-Driven Engineering - What Is Revolutionary, What Remains the Same? In *MODELS 2012, 15th International Conference on Model Driven Engineering Languages and Systems*, pages 692–708. Springer, October 2012.

[2] Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 476–491, Berlin, Heidelberg, 2005. Springer-Verlag.

[3] Clayton M. Christensen. *The innovator's dilemma: when new technologies cause great firms to fail*. Harvard Business School Press, Boston, Massachusetts, USA, 1997.

[4] Keith Davis. Methods for studying informal communication. *Journal of Communication*, 28(1):112–116, 1978.

[5] K.M. DeWalt and B.R. DeWalt. *Participant Observation: A Guide for Fieldworkers*. Anthropology / Ethnography. Rowman & Littlefield Pub Incorporated, 2002.

[6] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven Engineering Practices in Industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 633–642, New York, NY, USA, 2011. ACM.

[7] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, 2014.

[8] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.

[9] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture^{TM}: Practice and Promise*. Addison-Wesley Professional, Boston, MA, USA, 2005.

[10] A. Kuhn, G. C. Murphy, and C. A. Thompson. An Exploratory Study of Forces and Frictions affecting Large-Scale Model-Driven Development. *ArXiv e-prints*, July 2012.

[11] Grant Michelson and V. Suchitra Mouly. 'You Didn't Hear it From Us But...': Towards an Understanding of Rumour and Gossip in Organisations. *Australian Journal of Management*, 27(1 suppl):57–65, 2002.

[12] Lars Pareto, Peter Eriksson, and Staffan Ehnebom. Concern coverage in base station development: an empirical investigation. *Software and Systems Modeling*, 11(3):409–429, 2012.

[13] Colin Robson. *Real World Research: A Resource for Social Scientists and Practitioner-Researchers*. Regional Surveys of the World Series. Blackwell Publishers, 2002.

[14] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[15] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, 2012.

[16] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[17] C.B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[18] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. Industrial Adoption of Model-Driven Engineering – Are the Tools Really the Problem? In Ana Moreira and Bernhard Schaetz, editors, *MODELS 2013, 16th International Conference on Model Driven Engineering Languages and Systems*, Miami, USA, October 2013.

[19] Jon Whittle, Mark Rouncefield, and John Hutchinson. The state of practice in model-driven engineering. *IEEE Software*, 2013.